Project Number	AC026
Project Title	Secure Electronic MarketPlace for Europe SEMPER
Deliverable Security Class	Public
CEC Deliverable Number	AC026/SMP/CT2/DS/P/002/b1
Contractual Date of Delivery	August 31, 1996
Actual date of Delivery	September 24, 1996
Title of Deliverable	Deliverable D03 Basic Services, Architecture and Design
Contributing Workpackages	WP12, WP21, WP22, WP31
Nature of the Deliverable	Specification
Author(s)	N. Asokan (IBM), B. Baum-Waidner (R3), T. Himstedt (GMD), J.-H. Hoepman (CWI), G. Karjoth (IBM), D. Livas (EUROCOM), M. Mazoue (SEPT), A. Mueller (R3), M. Nassehi (IBM), T. Pedersen (Cryptomathic), J. L. Abad Peiro (IBM), B. Pfitzmann (Univ. Dortmund), S. Prins (KPN), K. Riede (ENC), L. Salvail (CWI), M. Schunter (Univ. Dortmund), M. Steiner (IBM), K. Tzelepis (Intracom), M. Waidner (IBM), D. Whinnett (Univ. Freiburg), R. Wolthuis (KPN)
Editor(s)	Cryptomathic
Keywords	Architecture, Electronic Commerce, Security

© SEMPER Consortium, represented by IBM France, La Gaude September 24, 1996.

SEMPER Consortium:

Basic Services, Architecture and Design

Deliverable D03 of ACTS Project AC026

Public Specification
September 24, 1996

© SEMPER Consortium, represented by IBM France, La Gaude September 24, 1996.

SEMPER is part of the European Commission's ACTS Programme (Advanced Communications Technologies and Services). Funding is provided by the partner organisations, the European Union, and the Swiss Federal Department for Education and Science.

The members of the SEMPER consortium are Cryptomathic (DK), DigiCash (NL), EUROCOM EXPERTISE (GR), Europay International (B), FINTEL (B), FOGRA Forschungsgesellschaft Druck (D), GMD - German National Research Center for Information Technology (D), IBM La Gaude (F), IBM Zurich Research Lab (CH), INTRACOM (GR), KPN Research (NL), Otto-Versand (D), RegionLink (B), r3 security engineering (CH), SEPT (F), Stichting Mathematisch Centrum / CWI (NL), Systems Synthesis Ltd. (UK), University of Freiburg (D), University of Dortmund (D) (having replaced University of Hildesheim (D)), Banksys (B), Banque Générale du Luxembourg (L) and Telekurs (CH) are associated with SEMPER.

For more information on SEMPER visit

<http://www.semper.org>

or contact

Gerard Lacoste
IBM La Gaude
Centre d'Etudes et Recherches
Le Plan du Bois
F-06610 La Gaude, France

Phone + 33 92 11 48 07
Fax +33 93 24 45 45
Mail lacoste@vnet.ibm.com

Michael Waidner
IBM Research Division
Zurich Research Laboratory
Säumerstrasse 4
CH-8803 Rüschlikon, Switzerland

Phone +41 1 724 8220
Fax +41 1 710 3608
Mail wmi@zurich.ibm.com

Management Summary

This report is the first publicly available technical deliverable of *SEMPER*, "Secure Electronic Marketplace for Europe." It describes the main results on the security architecture of the first year (September 1995- August 1996).

The main objective of *SEMPER* is to develop, implement, trial and evaluate an open security architecture for electronic commerce over open networks (e.g., the Internet). This will be done in several steps starting with a trial of basic services after the first year followed by the implementation and evaluation of more advanced trials in the following two years.

This report describes the initial version of the *SEMPER* architecture, the initial design and how it will be implemented. Details on the integration of this implementation in the trials can be found in Deliverable D04 ("Basic Services, Prototype and Internet Trial"), while more information on the trials can be found in Deliverable D05 ("First Year Surveys and Evaluation").

The architecture of *SEMPER* defines a set of service layers. The lowest layer contains supporting services (archiving, cryptographic functions, communication, user interface). On top of this, a transfer layer provides services for electronic payment and secure transfer of documents. The next layer, the exchange layer, combines transfers into fair exchanges, e.g., contract signing, certified mail. These services are used by the commerce layer to provide more complex business primitives, e.g., "send order" for filling in a pre-defined order template, signing it, and sending it to the business partner.

The architecture is open: New service modules can be integrated via generic API's (e.g., new payment systems by adapting them to a generic payment API). New business applications (e.g., an on-line auction) can be supported via dynamically loadable scripts that use the primitives of the commerce layer.

The initial design and implementation concentrates on what is absolutely necessary for the first trials, namely secure order, offer, payment, and delivery. Therefore only a subset of the *SEMPER* architecture is implemented initially. The first trials are based on the World-Wide Web. The design ensures that the prototype can be used with any web browser and server. It does not require specific security enhancements, like S-HTTP or SSL.

TABLE OF CONTENTS

1 Introduction **2**

2 Other Architectures **6**

2.1 Financial Services Technology Consortium (FSTC) *6*

2.2 Stanford Digital Library Project *9*

2.3 CommerceNet *10*

2.4 OpenMarket Inc. *11*

2.5 Sun's Java Electronic Commerce Framework *12*

2.6 Microsoft's Internet Security Framework *12*

2.7 Conclusions *13*

3 Architecture Framework **14**

3.1 Services, Interfaces and Protocols *14*

3.2 Service Blocks containing Managers and Modules *15*

3.3 Transaction Objects and Context *16*

3.4 Fault-Tolerance *17*

4 Model **19**

4.1 Players of the Electronic Marketplace *19*

4.2 Commerce as a Sequence of Exchanges and Transfers *21*

4.3 Exception and Dispute Handling *23*

5 Service Architecture Overview **25**

5.1 The Business Application *25*

5.2 The Commerce Layer *25*

5.3 The Exchange Layer *26*

5.4 The Transfer Layer *26*

5.5 Supporting Services *26*

6 Set-up, Registration and Certification **29**

6.1 Infrastructure *29*

6.2 Certificates *30*

6.3 User Registration and Certification *31*

7 Architecture of the Exchange Layer **35**

7.1 Containers 35

7.2 The Exchange Layer 37

8 The Commerce Layer **40**

8.1 The CLConnection Class 40

8.2 The Commerce Services 41

9 The Transfer Layer **43**

9.1 The Transfer Layer Services 43

9.2 Basic Classes 43

10 Service Blocks on the Transfer Layer **47**

10.1 The Payment Block 47

10.2 The Statement Block 51

10.3 The Certificate Block 55

11 Supporting Services **60**

11.1 The Communication Block 60

11.2 The Crypto Block 62

11.3 The Preference Block 65

11.4 The Access Control Block 69

11.5 The Archive Block 72

11.6 TINGUIN 76

12 Implementation Architecture **80**

12.1 Implementation Language 80

12.2 Integration into the World Wide Web 80

12.3 The SEMPER Application 81

12.4 Externalization 84

12.5 Routing/Addressing Between Managers and Modules 86

13 Summary and Outlook **88**

14 References **89**

15 Index **90**

16 Appendix A: Glossary **A.1**

17 Appendix B: Documentation of the API **A.5**

TABLE OF FIGURES

FIGURE 1 LAYERED SERVICE ARCHITECTURE 15
FIGURE 2 MANAGERS, MODULES AND ADAPTERS 15
FIGURE 3 ENTITIES (MANAGERS OR MODULES) AND THEIR TRANSACTIONS. 17
FIGURE 4 TRANSACTIONS AND SUB-TRANSACTIONS. 17
FIGURE 5 TRANSACTIONS WITH FAILURE. 18
FIGURE 6 RESTORING THE LAST RECOVERY POINT. 18
FIGURE 7 IF THE TRANSACTION IS NOT CONTINUED 18
FIGURE 8 PLAYERS OF AN ELECTRONIC MARKETPLACE 19
FIGURE 9 SIMPLE EXAMPLE OF A SEQUENCE OF EXCHANGES AND TRANSFERS 22
FIGURE 10 TRANSFERS (FIRST ROW) AND EXCHANGES OF PRIMITIVE TYPES 23
FIGURE 11 ARCHITECTURE OF *SEMPER* — OVERVIEW 25
FIGURE 12 SERVICE BLOCKS OF SUPPORTING AND TRANSFER SERVICES 28
FIGURE 13 AUTHORITIES AND FLOW OF INFORMATION 31
FIGURE 14: CONTAINER WITH GOODS 35
FIGURE 15: CONTAINER WITH GOODS 36
FIGURE 16: EXAMPLE INPUT FOR FAIR EXCHANGE 38
FIGURE 17: EXAMPLE OUTPUT OF FAIR EXCHANGE. 38
FIGURE 18: ESTABLISHING A CONNECTION IN THE COMMERCE LAYER 41
FIGURE 19 PAYMENT BLOCK ARCHITECTURE 48
FIGURE 20 NEGOTIATION PROTOCOL FOR SELECTION 50
FIGURE 21 NEGOTIATION PROTOCOL FOR CONFIRMATION 50
FIGURE 22: SIMPLE KEY EXCHANGE USED IN STATEMENT MODULE 53
FIGURE 23: SELECTING A KEY MUTUALLY AT RANDOM. 53
FIGURE 24: COMPLETE CLASS DIAGRAM 76
FIGURE 25: INTEGRATION INTO WWW 81

1 Introduction

As in a real marketplace, the main purpose of an electronic marketplace is to bring potential **sellers** and **buyers** together. Sellers **offer** their goods and buyers **order** these goods; together this is a two-party **negotiation**, often ending with a **contract.**
This may require that buyer and seller have some relations already established, e.g., to banks or government agencies. For instance, a buyer might only want to buy from sellers that are accredited with a well-known payment system provider (so that they can use a certain payment instrument) or from sellers approved by a consumer organisation. This requires **registration** and **certification**.
After the negotiation, the seller **delivers** the goods and the buyer **pays**; this is a two-party **(fair) exchange**. Goods should here be understood very generally. E.g., the buyer might receive a ticket that subsequently enables **conditional access** to certain services (such as subscription to a journal).
In this scenario third parties, such as **notaries** and **financial institutions**, may be needed in the actual business transaction.

An architecture covering all aspects of electronic commerce must not only be able to deal with the above generic services in an ideal situation. It should also be able to cope with technical problems as well as disputes between buyer and seller (in short, **exception** and **dispute handlers** are necessary).

The architecture described in this document covers the following typical situations which include the above scenario as a special case:

- **On-line purchase** A retailer accepts electronic orders and payments, based on digital or conventional catalogues, and delivers physical or digital goods (possibly copyright protected). An interesting example of this is when a user buys a ticket that can be used by a well-defined set of users (e.g., a family) to access a certain service for some time period or a certain number of times.
- **Subscriptions**: An organisation offers services on a subscription basis, e.g., subscription to news services, database services, or journals. Another similar scenario is subscription to an insurance.
- **Electronic Mall**: An organisation offers services for several service providers, ranging from *directory services* (“index”) over *content hosting* to *payment services.*
- **Auctions**: Users participate in an auction, maybe anonymously, and with the usual fairness requirements. This situation is atypical as more than one buyer will participate
- **Secure communication**: Secure transfer of electronic documents offering confidentiality, non-repudiation services, etc. This also includes secure electronic mail.
- **Contract Signing**: Two or more parties exchange signed copies of the same statement.

There will be several different implementations of the services offered by the marketplace fulfilling different functional requirements and different security requirements, and following different business models. For instance, payment can be

designed in many different ways (e.g., pre-paid or post-paid), and for each design many different actual implementations may be available. Another example is registration which may be performed by different organisations using different registration procedures and different types of certificates. The *SEMPER* architecture is open in that different designs and implementations may be integrated provided they have a suitable API.

Basic Services

The definition of the *SEMPER* architecture is based on the above scenarios envisaged to be useful in an marketplace as well as the functionality required by the three service providers EUROCOM, FOGRA and Otto Versand.

For the initial trials these service providers identified five basic security requirements: Authentication (identification), Integrity, Signature, Payment, and Confidentiality.

Authentication

Authentication (reliable verification of identity) is frequently required by both the merchant and the customer at various stages in electronic commerce. The first example of this requirement arises at the initial point of contact between merchant and customer, the electronic shop-front. The customers entering this electronic shopping environment may need confirmation, that he/she is genuinely in contact with the company he/she has selected and not an electronic impostor.

Authentication of the customer to the merchant is dependent on the specifics of each customer merchant relationship. Otto for example does not need to know the identity of a customer until an order is placed, while FOGRA or EUROCOM may restrict access to certain information to only those customers that they have authenticated. EUROCOM, for example, frequently deals with students attending prepaid courses, who should only be granted access to their services for the duration of that particular course. FOGRA requires controlled access because they offer different conditions for members and non-members.

Integrity Protection

Two of the service providers will be delivering their products on-line and it is essential that the customer receive the correct service and that it be received intact. Both EUROCOM and FOGRA require confirmation that their services have been delivered to the customer.

Transfer of Signed Statements

The requirement for certification of the identity of merchant and customer mentioned above applies in particular to three aspects of the electronic transaction, i.e. offer, order and receipt. At each of these points in an electronic transaction both merchant and customer may require a secure and non-repudiable electronic exchange of information, e.g. the customer must be able to treat the merchant's offer as binding, the merchant must be able to regard the customer's order as binding and (as already mentioned) there must be proof of delivery/receipt of the "product".

Payment

The trials should support traditional payments by means of invoice and bank transfer, but also include the secure transfer of payment card data. In addition, there should be the opportunity to test implementations of electronic money. All service providers in the trial require electronic invoices/receipts for their customers, regardless of whether they prefer to settle their accounts on-line or off-line.

Confidentiality
All the service providers perceive a potential customer requirement for confidentiality. For example customers seeking information or advice in respect of competitively sensitive areas of business, where the client might not necessarily wish outsiders to be aware of an active business interest in the particular field.

Overview

This report describes the *SEMPER* architecture and the initial implementation supporting the services needed in the first trials. The architecture is covered by Chapter 2 through 7 defines the architecture while Chapter 8 through 12 give an overview of the implementation.

Chapter 2 takes a brief look at other projects dealing with general architectures for electronic commerce and discusses their relevance to *SEMPER*.

Chapter 3 then describes the architecture framework. The central issue introduced here is service blocks. A service block consists of a manager and a number of modules. The manager provides the required services using one of the modules (e.g., the payment block will use existing payment systems as modules). A manager provides a generic interface such that several modules (possibly through an adapter) can be used in *SEMPER*. This is the basis of the independence of specific implementations of the modules — one of the key points of *SEMPER*.

Chapter 4 presents the model of electronic commerce upon which the *SEMPER* architecture will be developed. Among other issues this chapter describes the actors in the electronic marketplace: seller, buyer and third parties enabling the required services. This chapter also discusses how electronic purchases can be considered a sequence of transfers and exchanges. A typical example is when digital goods are delivered in exchange of a payment.

SEMPER has adopted a layered architecture which is described in Chapter 5. On top we have the business applications which use security services provided by the commerce layer. These services are implemented using services for secure transfer and exchange as mentioned above. These transfer and exchange services are implemented using service blocks and each layer will contain a number of these.

Chapter 6 presents some principles for setting up *SEMPER* in such a way that users can trust it. This chapter also deals with the registration and certification of users.

Chapter 7 discusses containers, which are used to unify the concepts of transfer and exchange. The first implementation of the *SEMPER* architecture does not contain exchanges, but this chapters presents the initial ideas regarding the exchange layer.

The following chapters describe in a top-down manner the design and implementation of the various layers of the architecture starting with the commerce layer and transfer manager in Chapter 8 and Chapter 9, respectively.

Chapter 10 describes the three blocks on the transfer layer: payment, statement (for secure transfer of information) and certificate. The payment block provides services needed for payment while the statement block is used for securing other information. The certificate block is used for registration as well as handling of certificates.

Chapter 11 describes the service blocks needed to support the security relevant services of the transfer layer (e.g., cryptographic tools and services for archiving and communication).

Chapter 12 briefly describes how the architecture will be integrated in the business applications in the trials.

Appendix A contains a glossary defining the most frequently used terms.

The architecture is implemented in JAVA and documented using JAVADOC. Appendix B gives guidelines for where and how to retrieve this documentation.

2 Other Architectures

Electronic commerce has received considerable attention in recent years as the development of security techniques has allowed not only secure transfer of information but also secure and practical electronic payments systems. This chapter describes the most interesting projects on electronic commerce. A short "summary" subsection in each section attempts to highlight the relevant aspects of the respective projects from the point of view of *SEMPER*.

2.1 Financial Services Technology Consortium (FSTC)

FSTC is a consortium consisting primarily of US banks with contributions from other financial institutions, research organisations and universities (see [FSTC]) Its goal is to sponsor research and development relevant to the financial industry. Currently, it sponsors four projects:

- **Electronic Check**: use of a personal check book device to write out cheques and transport them (over insecure networks) to payee or to a bank**.**
- **Electronic Commerce**: development of a full architecture; building of a testbed and prototype applications.
- **Check Image Exchange**: standardisation of cheque image representations.
- **Fraud Prevention and Control**: use of technology in fraud prevention and control.

Of these, the latter two do not have publicly accessible information on the technical details. They also appear to be in the planning stages.

2.1.1 The Electronic Check Project

This project aims to provide an electronic payment/deposit mechanism that can be integrated into the existing banking infrastructure. The *E-check* is modelled after the paper cheque. It will be created by a tamper-proof hardware device called the *electronic check book* which will house a private key of a unique keypair. In addition, if possible, it will also keep a register of cheques signed, endorsed, and issued. The E-check may be implemented as a tamper-resistant PCMCIA card, or an ISO format smartcard, or in the form of special-purpose cryptographic hardware processors (for large-scale use).

Using attribute parameters and routing information, an electronic check can be made to behave like a traveller's cheque, certified cheque etc.

A certification hierarchy will be used to issue "electronic check certificates" to be used in authentication. The project therefore assumes an appropriate infrastructure. A goal of the project is to find out requirements for such an infrastructure.

The project seeks to develop an API for electronic cheques and hopes that it will be used by a wide variety of financial application developers. It will also make a prototype implementation which will provide the services of this API. A two layer architecture is implied; the upper layer provides high-level functions broken up into the following categories:

- Management, (Bank)

- Check clearing, (Bank)
- Check writing, (Payer)
- Reconciliation, (Payer)
- Check acceptance and endorsement, (Payee)

The lower layer exports functions for manipulating the e-check, such as:

- Write (create a cheque and bind to a document if necessary),
- Co-sign,
- Verify,
- Endorse,
- Register read,
- Register entry

2.1.2 The Electronic Commerce Project

The goal of this project is to design and demonstrate an infrastructure for electronic commerce. While the project is still in the outline stage, its perspective is from the banks' point-of-view and its emphasis is on facilitating the inter-bank clearance procedures. Customers will use the Internet to communicate with the bank and with each other, using payment instruments like the E-check and a payment protocol to be chosen. An **Electronic Payment Handler (EPH)** at each bank will process these transactions. EPHs will use existing inter-bank clearance mechanisms. If both banks have EPHs, then "EPH Payment Servers" on both banks could directly communicate over the Internet and resolve the transaction based on a prior bilateral clearance arrangement.

The tasks envisaged for the project are the identification of requirements for an appropriate **Network Payment Protocol (NPP),** the design of EPH, and extensions to the ATM and Point of Sale networks to support the direct clearance of small payments within a short time.

Many requirements for the payment system are identified. Apart from basic requirements about correctness, security, FSTC also identifies "accessibility" from a wide variety of platforms, scalability (in terms of the volume and the range in the amount of a transaction) and the use of Internet as requirements.

Network Payment Protocol

The project will identify the requirements for the NPP and then choose the best available candidate. There are two components to the specification of a suitable NPP: message formats and transaction flows.

Message Formats

Suitable formats to express payment instructions, authorisation requests/response, status request/response, etc. need to be defined. In particular, the payment instruction "*is likely to contain a superset of the data elements included to a lesser extent in each of the other message types"* [FSTCa]. In other words, the need for cross-reference of information is implied. In addition, message formats should support the specification of various security enhancements. Security enhancements may be implied by the characteristics of the payment or explicitly specified.

Transaction Flows

These should include provisions for replay detection, handling of error situations etc. The rest of the requirements are essentially the same as those for message formats.

Future extensions to the NPP should support “non-payment data transmission”, other cash management transactions (like account statements), additional security mechanisms (e.g. biometrics) and requirements (e.g. anonymity).

Electronic Payments Handler

The EPH is the meeting point for both the customers on public networks and the banks' back-end systems. It has four components.

EPH Frontend

This is meant to be the “first line of defence.” No additional details of its functions are available. Presumably, it serves the same purpose as a firewall.

EPH Server

When presented with a payment request, the EPH server's task is to perform basic validation of the request, select the appropriate mechanism (based on user preferences, amount, timing etc.), formatting of appropriate messages to be passed to the payment system, and the handling of status requests during the payment process.

EPH Certification Server

Each bank is responsible for issuing and revoking certificates for its customers and for verifying received certificates.

EPH Payments Server

By default, payments will be processed by existing inter-bank channels. But when both (and later all) banks involved in a payment have an EPH, they can resolve the payment directly based on bilateral agreements. This process is carried out by the EPH Payments Server at each end. A separate protocol and necessary message formats need to be specified for this communication.

2.1.3 Summary

The Electronic Commerce Project is intended to enable banks and other financial institutions to provide support for electronic payment among their customers. Customers can use a variety of payment tools including the Electronic Check. It appears that these efforts are complimentary to the goals of *SEMPER* since FSTC seems to focus on an architecture for payments rather than for a general electronic marketplace. The *SEMPER* payment manager can treat the E-check as yet another payment instrument and the inter-bank clearance mechanism is outside the scope of *SEMPER*. It should also be noted that *SEMPER* has an official liaison with the FSTC Electronic Commerce Project.

2.2 Stanford Digital Library Project

The Stanford Digital libraries project (see [SDLP]) aims to develop the infrastructure for a universal integrated digital library. Originally, they have sub-projects in several areas including:

- **Information interfaces**: Resolving information in multiple formats, communication about documents etc.
- **Information finding** : Resource discovery mechanisms
- **Economic perspectives**: Payment mechanisms for document access
- **Infrastructure and models**: Meta-information about document and repositories, visualisation of results
- **Agent technology** : The scope for using agents throughout the system

Here we focus on one such sub-project. However, they have other sub-projects of interest, for example in copyright protection.

2.2.1 InterPay Architecture

As part of the "Economic Perspectives" thrust, they have developed an architecture called *InterPay* that embodies the economic aspects of the for-pay (i.e. not free of cost) digital library. The goal is to manage multiple payment mechanisms and policies with a coherent approach. The architecture consists of three "layers": the *services layer*, the *payment policy layer*, and the *payment mechanism layer*.

A client interacts with a service provider over the services layer and chooses what items are to be purchased. The server then invokes a "collection agent" in the payment policy layer with information about the payment for the selected items. The client invokes a "payment agent" in the payment policy layer with similar information. These two agents engage in a negotiation and agree on the specific payment mechanism to be used. The payment agent then invokes a "payment capability" corresponding to the chosen mechanism in the payment mechanisms layer with all the information necessary to make the payment using that mechanism. The collection agent invokes a similar "collection capability." The two capabilities use the mechanism to settle the payment.

The layering is not strict, in that *(a)* inter-layer communication between two different stacks is possible and is used to make the system flexible and *(b)* lower layer entities (e.g. payment agents) export interfaces directly to the user.

From coding experience, they have decided that the distinction between payment agents and collection agents is unnecessary and a unified "financial agent" could be used instead. Security issues have not been addressed seriously. They assume that the underlying object system used in the implementation will/should provide the necessary security services. The fair exchange problem and trust issues have not been addressed either. However these appear to be on-going work (e.g. see the *Making Trust Explicit in Distributed Commerce Transactions* document).

Recently, they have begun the second phase of this project called *InterPay II* in which they are developing a common API for payment systems (similar to the *SEMPER* payment service block) and plan to address payment-related transactions such as offer, order, and delivery.

2.2.2 Summary

This project seems to be relevant to *SEMPER* as it addresses advanced research issues and not merely secure payments, it seems to be more focused and active, and the work on InterPay and on-going work on trust issues is particularly relevant to *SEMPER*. There seems to be very little that is specific about digital libraries at least in the economic perspectives thrust. It would make more sense for digital libraries to be a participant in a more general electronic marketplace.

2.3 CommerceNet

CommerceNet (see [CN]) is a consortium with a goal to create an electronic marketplace. It has several working groups which have some pilot projects under way. CommerceNet has provided SHTTP as an "enabling" mechanism to base the pilot projects on. This protocol allows communication using HTTP with a combination of prescribed security enhancements applied to the messages. It is also possible for two entities to "negotiate" security requirements for their communication or for an entity to unilaterally advertise such requirements for any communication with it.

CommerceNet projects are somewhat decentralised. There is no single "big project." Instead, there are various small scale pilots sponsored by the different working groups and carried out by a group of members of the consortium. Hence there is some overlap and some degree of incoherence.

2.3.1 Details about Pilot Projects

Electronic Commerce Pilot - A trial electronic market is to be constructed using proprietary technology of OpenMarket Inc. No technical details are available in the pilot proposal. It does not look as if SHTTP is used. They have a Payment Server to facilitate commerce between stores built with their store building package. It appears to be a highly centralised architecture.

Trusted Software Distribution Pilot - Procedures to establish the veracity of the origin of software will be established; a trusted software certifying authority will then issue a certificate to the submittor which can then be distributed with the software package.

Key Safeguarding Pilot - This will establish a mechanism for recovering secret data, in particular keys. During key generation, a data recovery centre (DRC) is contacted. After authentication, the DRC will provide the information necessary for the key generation software to produce a *Key Recovery File* which can then be safely backed up. For recovery, the DRC is contacted again. It is not clear how authentication takes place at this point (they do indicate that this process is **not** manual!). Multiple DRCs can be used.

Payment Technology Pilot - This seems to have the same scope as the Electronic Commerce Pilot. A payment protocol will be built, likely on top of SHTTP. A server system provided by CyberCash will be located on the Internet and the banking networks and function as a go-between between the customers and the bank(s) .

Value/Effectiveness Measurement Pilot - This pilot attempts to develop a framework of metrics to measure the effectiveness/impact of electronic commerce. The

focus is on what sort of instrumentation can be built into applications in order to make these measurements. It would be interesting to see the final white paper from this project.

In addition, there are a handful of pilot projects on deploying catalogues and improving their usability. A working group called the **EDI working group** has been established. Its purpose is to define an architecture that links buyers, sellers and service providers through the Internet as well as proprietary networks; develop a set of electronic commerce services. They don't seem to have any pilots yet (at least none advertised). As indicated, some of the pilot projects mentioned above have (or should have) already been completed according to the pilot proposals available at the CommerceNet WWW site. However, progress reports or final white papers do not appear to be readily available. More up to date information can be gleaned from the CommerceNet Mail Archives.

A joint project between CommerceNet and the W3 Consortium to develop a Payment Negotiation Protocol has been announced.

2.3.2 Summary

CommerceNet projects are rather loosely organised. Specific projects that may be of interest have been identified in the previous section. Of particular interest to *SEMPER* is the approach used in the definition of SHTTP and the forthcoming payment negotiation protocol.

2.4 OpenMarket Inc.

OpenMarket Inc. (see [OM]) has had a payment switch operational for some time and has built an integrated commerce infrastructure around this payment switch. The infrastructure is based entirely on the World-Wide Web.

2.4.1 Features of the Integrated Environment

In addition to payment processing itself, the integrated environment allows for, among other things:

Order Taking: The “digital offers” in a shop are in the form of encoded Uniform Resource Locators (URLs) that contain a pointer to the payment switch and enough information about the goods, prices, and authenticity of the offer. A buyer can choose to buy an offer right away or collect items into a “shopping cart.”

Order Fulfilment: “Digital receipts”, similar to the digital offers, are URLs given to the buyer by the payment switch. They encode information for buyer authentication, time limits, and for product identification. This can serve as receipts for the purchase, a means to access the goods (if they were “soft” goods), or a means to refer back to a past purchase.

The payment itself is by means of a credit card number. Depending on the configuration, the payment switch will either use any existing authentication mechanisms provided by the underlying protocol or perform additional exchanges (e.g. a challenge/response interaction) with the potential buyer. Once authenticated, the switch will check the user’s credit card authorisation using the appropriate credit card network.

There are tools for creating the digital offer URLs and for interpreting the digital receipt URLs.

2.4.2 Summary

The features that make the OpenMarket environment interesting are that (a) it works with existing protocols and tools in the World-Wide Web infrastructure but can also take advantage of secure protocols and tools when they come to pass, and (b) it comes with a wide range of support tools that would ease the initial effort in setting up a shop in their integrated environment. *SEMPER* trials based on the WWW may benefit from a closer examination of the techniques used by OpenMarket. On the downside, it is a highly centralised architecture.

GC Tech (see [GCT]), a software technology company in France, also uses a similar approach in their secure intermediation system.

2.5 Sun's Java Electronic Commerce Framework

Sun has recently announced a "Java Electronic Commerce Framework (JECF)" (see [JECF]). JECF is intended to be implemented as a software "*virtual point-of-sale device*" that can be accessed and used by "web browsers and other Java-enabled environments" [JECFa]. The general scope of JECF is roughly the same as that of the *SEMPER* payment service block; however, it "may be extended to provide other types of financial services." Even for a limited scope, many of the supporting services (archival, access-control scheme etc.) required for a more complete solution have to be designed and built anyway.

JECF has introduced the notion of cassettes as a means to add new applications to the framework (e.g., new payment systems can be added using cassette).

Another notable aspect about JECF is the proposed access-control scheme. Using the facilities available in the Java language, the JECF proposal outlines an elegant access-control scheme.

On the other hand, the payment API defined in JECF is not very complete. In particular, only the services required at the client side seems to have been taken into consideration.

2.5.1 Summary

It appears that JECF was announced at an early design stage. Based on the very limited information publicly available at present, it is difficult to make any predictions about how *SEMPER* and JECF can fit together, but it seems that JECF aims at achieving some of the same goals as *SEMPER*.

2.6 Microsoft's Internet Security Framework

Microsoft's *Internet Security Framework* (see [MISF]) is a collection of a comprehensive set of security technologies that can be used to enable various applications in a secure manner, including electronic commerce. It consists of the following pieces (some of them are existing or proposed standards and some are new proposals):

- **CryptoAPI** is a generic API definition of cryptographic services. In scope, it is similar to the *SEMPER* crypto block and the client side of the certificate block. As with *SEMPER*, the emphasis is on being able to plug-in any cryptographic module (called "Cryptographic Service Provider" in their terminology) to applications. The CryptoAPI forms the basis of the rest of the framework.
- **Certificate Server** is a certification authority that can make use of a variety of certificate algorithms and support different, configurable, policies on certificate management.
- **Wallet** is the repository for all the credential information that an entity possesses. It is similar to a secure archive in *SEMPER* terms. The information in the wallet will be made available to the applications that need them subject to the local access-control policy. Entities can use the proposed **PFX** (Personal Information Exchange) protocol to transfer such credential (or other sensitive information) from one platform to another.
- **Code Signing** - **Authenticode** is a mechanism where software distributed electronically can be cryptographically signed by the publisher and can be verified by the users.
- **SET** is a joint proposal by Mastercard and Visa to secure credit-card transactions on the Internet. Unlike the *SEMPER* payment block, the Microsoft framework does not yet address the possibility of using payment mechanisms other than SET.
- **Secure Channel Services** provide point-to-point confidential communications channels between two participants. **PCT** and **SSL** will be used to provide this service. One can view this as a restricted version of the secure channel services provided by the *SEMPER* transfer layer. PCT (Private Communication Technology) is a protocol that provides a confidential, authenticated, connection-oriented communication channel. SSL (Secure Sockets Layer) is a proposal by Netscape to address the same problem. In addition, on-going work by the IEEE Transport Layer Security working-group will be supported by the framework.

2.6.1 Summary

The proposed framework identifies all the necessary pieces of technology required for a secure electronic marketplace. *SEMPER* differs from this framework in two ways:

- each service block in the *SEMPER* architecture has a similar structure which arises out of the same goal of defining a uniform way of using different modules that provide the same service. The Microsoft framework achieves this in some areas (e.g. CryptoAPI) but not others (e.g. payment systems).
- Since *SEMPER* provides an integrated framework it is relatively easy to develop scenario-specific applications (called "business applications"). The Microsoft framework is less integrated in this respect even though the services eventually provided are roughly the same

2.7 Conclusions

Of the projects surveyed, none appears to have the intended scope of *SEMPER*. In addition to these six projects, a number of other projects/solutions have been announced. They are not addressed here because they do not constitute complete architectures but merely function as payment switches.

3 Architecture Framework

This chapter discusses general notions needed to define the *SEMPER* architecture.

3.1 Services, Interfaces and Protocols

SEMPER defines a layered service architecture. Since we make a very informal description, we do not need a formal semantics of "layer" and "service". But in principle we use these and related terms in the sense of the ISO OSI model [ISO 7498]:

The active elements in such an architecture are called **entities**. Each entity resides in one device, where **device** means a physical object, such as a personal computer. The term **local** means "in the same device". Usually, a device belongs to the player (or set of players) who relies on it.

A **layer** is a (virtual) collection of entities in all devices that perform functions of a similar degree of abstraction. More basic functions are collected in lower layers, and higher layers perform more complex functions using the services provided by the lower layers. Layers are usually numbered 1, 2, …, starting with the lowest one. The interface between two layers is called a **layer interface**. It is the union of all interfaces of the entities of this layer. Entities within the same layer providing the same service on different devices are called **peer entities**.

The joint functionality that a layer offers at its upper interface, i.e., to the higher layers, is called the **services** of this layer. A layer can offer several services.

An entity can access the services of any lower layer at a **service access point (SAP)**, see Figure 1. A **service interface** of a service is a subset of the layer interface, consisting of service access points and a syntactical definition of the **service primitives** that can be exchanged via these service access points. An entity accessing (or using) a service is called an **invoking entity**.

A **service** is described by characterising the possible behaviour on the whole service interface, i.e., how inputs and outputs at the different service access points are related, but *without* describing how these relations are implemented.

A **protocol** describes how a service is provided by means of interactions of peer entities via lower layer services. Typically, the same service can be provided by many different protocols, and some of these different implementations may even be present in the same system. A protocol that implements services of Layer i is based on the services of the lower layers 1,...,i-1 and can also invoke other services of its own layer. Thus access is not restricted to the layer immediately below, even though protocols will mainly use services of the layer immediately below.

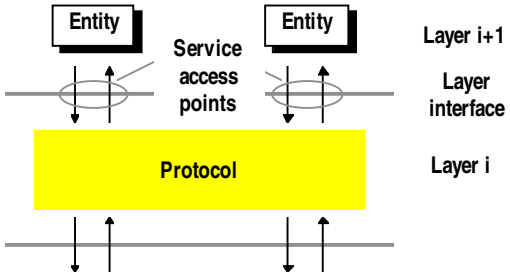


Figure 1 Layered service architecture

3.2 Service Blocks containing Managers and Modules

Generic services are provided by **service managers** and several **service modules** (see Figure 2). The union of a manager and its modules is called a **service block**.

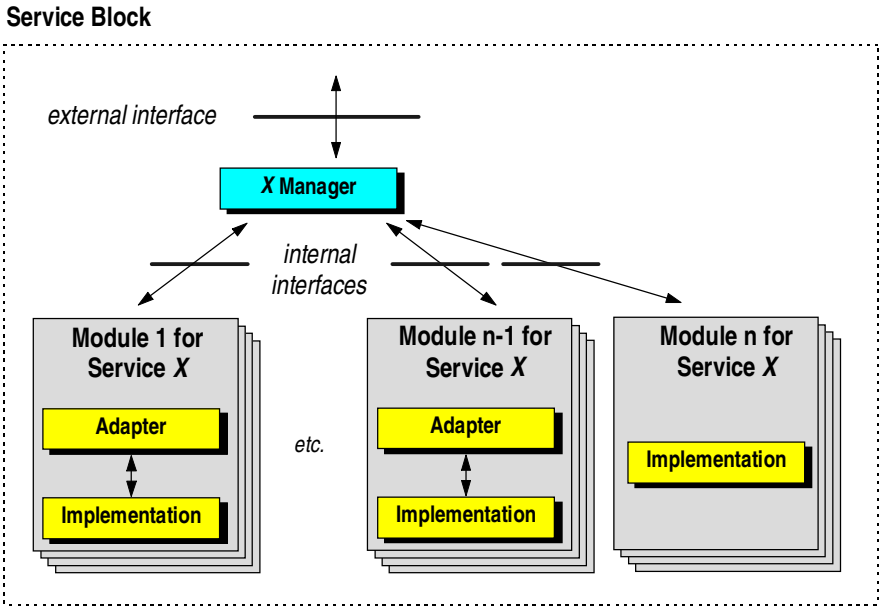


Figure 2 Managers, modules and adapters

The last module was written specifically for SEMPER and needs no adapter.

The **service block** provides the generic, unified service, e.g., service = “payment”. We call the interface supported by the entities of the block the **external interface** of the service block.

These external interfaces provide a **generic service**. The generic service is based on a model of the service which should cover a broad range of protocols implementing this service, i.e., we will have one generic interface for a whole class of services.

For instance, the external interface of the payment manager is based on a *generic payment service* that will cover all kinds of *payment protocols* (or at most at a small number of generic payment services, one for each payment model). Note that this does *not* mean that we will support only a few specific payment protocols, but that the interface definition is so general that *any* reasonable payment protocol can be accessed via that interface. If, e.g., company *XYZ* comes up with a new payment system *abc*, all they would have to do in order to link *abc* into our architecture is to map the service

interface of *abc* to the internal payment interface. This guarantees the desired openness of the marketplace.

A **service manager** provides a common interface to several modules together with methods for negotiation and selection of an appropriate module.

A **service module** corresponds to a protocol implementing the service, i.e., it more or less implements one entity of such a protocol. Its interface to the service manager is called an **internal interface** of this service. Each generic service model may define several internal interfaces based on the **type** of the expected service of its modules.

Having several modules per service allows different types of service and different protocol implementations by different manufacturers. Service modules are said to be of the same type if their behaviour at the internal interface is the same. Examples of types of internal payment interfaces are “cash-like”, and “credit card-like”. Modules could be “SET”, and “e-cash” where “SET” implement the credit-card model whereas “e-cash” implements the cash-like model.

As *SEMPER* wants to build on existing products as far as possible, we cannot assume that they all fit the same internal interfaces originally. Therefore, the interface of an existing implementation is enhanced by a **service adapter** so that the resulting module supports the required service.

The *SEMPER* architecture describes a fixed set of service managers. The set of service modules is not fixed, i.e., service modules can be dynamically attached to managers.

3.3 Transaction Objects and Context

A protocol execution may require the interacting entities to agree on some parameters such as type and quality of service to be used. Two peer payment service blocks may need to agree on the payment system to be used. The sub-protocol that negotiates this agreement is called **negotiation**. Results of negotiations are stored in the so called **context** of the transaction. Context is a data structure local to one entity that contains the results of user-defined preferences, configuration and the result of negotiation. Note that the negotiation is usually not a negotiation of actual business data from the users.

Services of a service block in *SEMPER* can either be used directly or through a so called transaction object. A transaction object is managed by the manager of the service block and when instantiated it defines a context for all services (possibly by negotiating with the peer entity - see Figure 3). Thus the advantage of using the services through a the transaction object is that many parameters of the service is fixed by the context. The manager is notified during instantiation of the new transaction object and keeps track of all ongoing transactions of the entity.

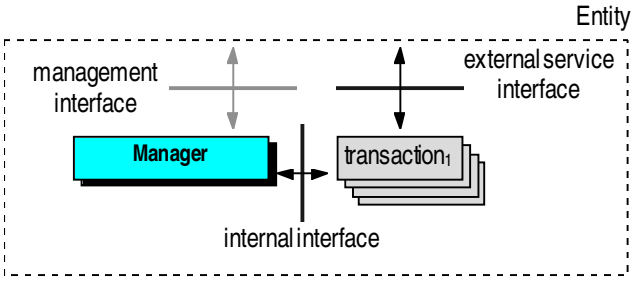


Figure 3 Entities (managers or modules) and their transactions.

To prevent re-negotiating parameters which were already negotiated, the result of a negotiation can be stored in a **profile** for later use.

3.4 Fault-Tolerance

We now describe the concept how service blocks can implement fault-tolerant **transactions** in the object-oriented model [LeAn90]. A *SEMPER* transaction is a sequence of method invocations which implement a state transition from one consistent state to another where the **state** of a transaction is the set of all its objects which are stored persistently. A payment transaction should, for example, ensure that it either ends in a state before or after a payment, i.e., if it is interrupted during the execution of a payment protocol, it should either complete the protocol or clean-up by undo the effects of the protocol. To achieve this property in the presence of faults, every transaction must implement so called recovery mechanisms to get into a consistent state after the failure.

Since protocols of a transaction may use other services of other transactions, transactions are ordered hierarchically. The transaction started by another transaction is called its **sub-transaction** (see Figure 4). For example, making a payment may involve transactions for the actual payment and for a transfer of a signed statement.

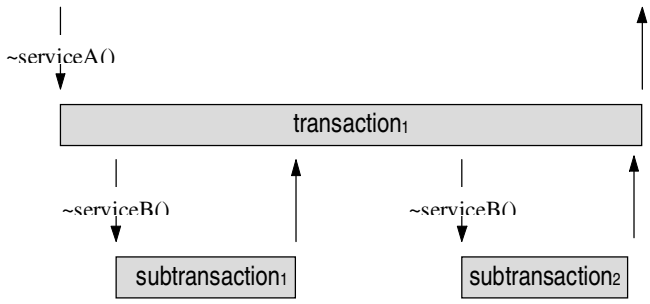


Figure 4 Transactions and sub-transactions.

During execution or when starting sub-transactions, each transaction stores persistent states at so called recovery points.
After a crash as in Figure 5, all ongoing transactions need to be recovered, i.e., the recovery procedure should either “roll-back” into the last consistent state or redo parts of the transaction to reach the next consistent state. The recovery is performed top-down: The topmost manager recovers its transactions by re-instantiating the transaction objects and calling the method which has been interrupted. The transaction

restores the last recovery point which may involve instantiating sub-transactions and either undoes all actions and reports a failure or continues (see Figure 6 and Figure 7).

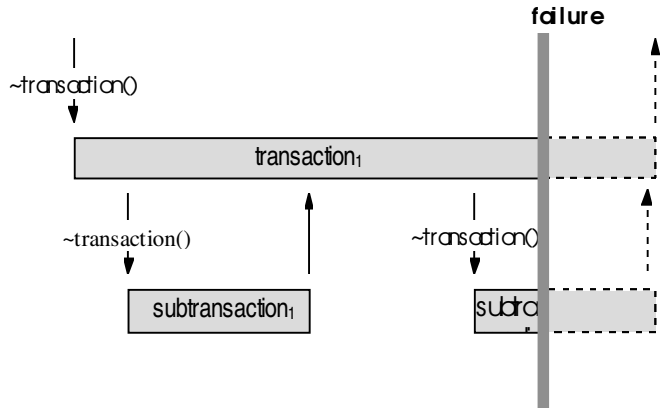


Figure 5 Transactions with failure.

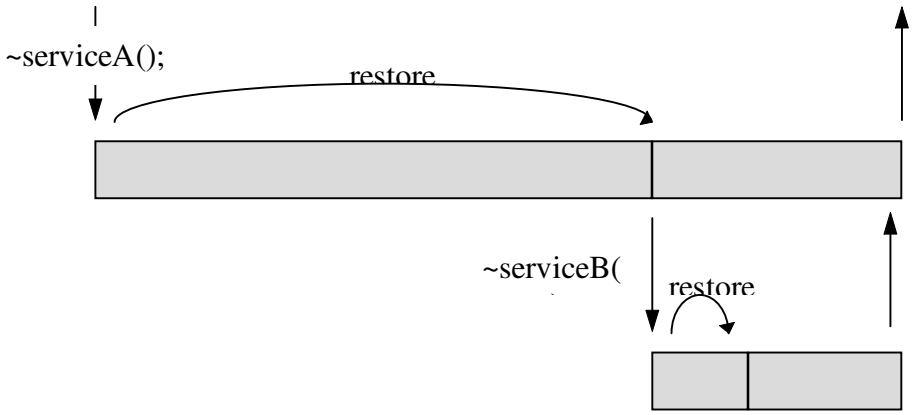


Figure 6 Restoring the last recovery point.

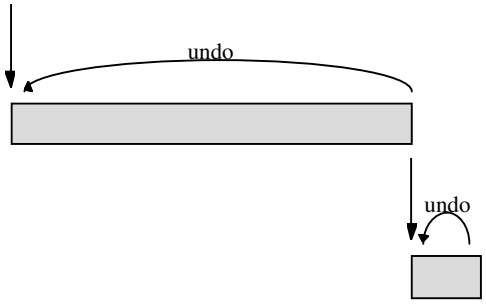


Figure 7 If the transaction is not continued, all persistent changes which would represent an inconsistent state should be undone.

Since recovery of each transaction depends on the details of the algorithm, the recovery is implemented by each transaction object which must makes sure that all objects needed for recovery are stored persistently in the archive to enable consistent recovery.

4 Model

This chapter presents a model for electronic commerce, by introducing the key players and the properties, that we want to focus on.

4.1 Players of the Electronic Marketplace

The services listed in the introduction involve several **players**, i.e., types of assumed real-world people or bodies participating in the marketplace. Figure 8 lists the essential ones in an electronic marketplace.

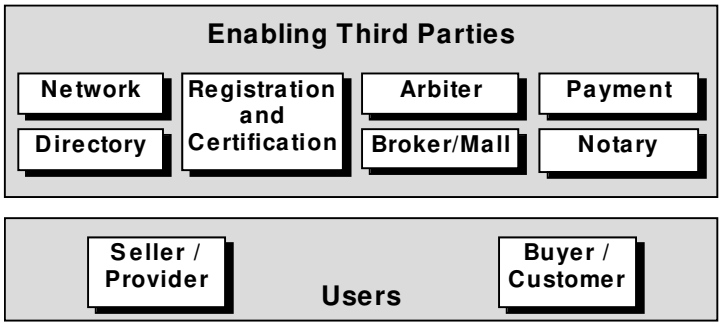


Figure 8 Players of an electronic marketplace

From the perspective of a single business session (e.g., a purchase), the participants in a marketplace are acting as

- **users** (of the marketplace) i.e., buyers and sellers, also called customers and providers, and
- **third parties**, who enable the session between the users, e.g., network and payment system providers.

Note that these are just convenient names of ***roles*** (i.e., we use "player" as a synonym of "role", the words "participant", "party", and "person" are left for less restricted usage). There is no partitioning of all real participants into such classes: A bank might act as the *provider* of a home banking service at one time, and at the next time as a *third party* that guarantees a payment of the customer.

Figure 8 lists those enabling third parties we expect to be present on most electronic marketplaces. We do not use the term *Trusted Third Party (TTP)* since:

- Most authors do not distinguish between *TTPs* and *TTP Services.* Any party can provide the latter, but whether a specific provider of such services is a TTP, i.e., trusted, is completely up to each individual user to decide. There is no such thing as a TTP in an objective sense.
- The main objective of security protocols is to limit the trust necessary in other parties. In fact, most services provided by so-called TTPs can be implemented such that not much trust is required at all.[1]

[1] First, most services can be implemented using several third parties such that one honest third party among all of them is sufficient, i.e., trust is *distributed*. Secondly, TTPs can be made *accountable*. Sometimes, this has to be based on the physical world. For instance, the certification

- Conversely, the term TTP suggests that a user does not need to trust those third parties not called TTP. Unfortunately this is not true: For instance, a user has almost no chance to verify that her PC works correctly, i.e., she has to trust the manufacturer of this device. But usually nobody calls manufacturers of computing equipment TTPs.

The purpose of the third parties mentioned in Figure 8 is:

Network: The network describes the providers of communication infrastructure used on the electronic marketplace.

Directory: A directory is a (possibly distributed) database that maps a name of a participant to a directory entry and its attributes such as the name holder's description, addresses, certified public keys, etc.

One might consider physical *phone books* the analogue of directory services. Relevant standards are, e.g., *X.500* and the Internet *DNS*.

Registration and Certification: A *registration and certification authority* (abbreviated RA and CA, respectively) links digital and real identities of a user who wants to register. For this, it verifies the user's real identity (registration) and certifies a public key of the user (certification), so that the user can now act under the digital identity.

One might consider physical *passports* the analogue of certificates. The most important standards for certification schemes are *PKCS #6*, *X509* [X509], and the *UN/EDIFACT Certificates* [EDIFACT].

Arbiter: An *arbiter* evaluates the meaning of digital evidence based on protocol specific *decision procedures*. The decision procedure of a protocol defines which messages serve as evidence, and defines their semantics, i.e., what they are evidence for.

Note that properties like *non-repudiation* or *disputability* do not make any sense without well-defined decision procedures. Since dishonest behaviour cannot be prevented in many cases, requirements on who would win a dispute in a certain situation is often the only way to express specific security properties.

Ideally, everybody should be able to evaluate a decision procedure, i.e., the arbiter's decisions should be predictable. One might consider an *expert witness* the analogue of an *arbiter*.

Broker: Like a *directory*, a *broker* is a database of entities of the marketplace, but it enables more intelligent requests, e.g., a characterisation of a service requested.

One might consider physical *Yellow Pages* as being the analogue of a *broker*.

---

of public keys for signatures can be made disputable if a written contract between CA and user is required, and the user's public key is included in that contract. The CA can still issue false certificates, but in case of a dispute the user can prove that the CA has certified the wrong key, and thus the user is not held responsible for the CA's fault.

Mall: A *mall* provides a common interface to several service providers. This includes

- *service gateways* that translate and redirect requests from buyers to sellers and the responses back to the buyers,
- *content hosting*, where the mall provides the services on behalf of the sellers.

One might consider physical *shopping centres* the analogue of a *mall*.

Payment: This means the *payment system provider* as well as all other financial institutions (*issuing banks, acquiring banks*) that participate.

Notary: Notaries will provide or support trustee services like

- time-stamping,
- contract signing,
- fair exchange of values, and
- long term archiving.

If necessary, we will add specific third parties for these services later.

4.2 Commerce as a Sequence of Exchanges and Transfers

Our model describes business sessions in terms of sequences of *transfers* and *exchanges* (similar to the *dialogues* of interactive EDI).

In a **transfer**, one party sends an item, called a *container*, to one or more other parties. The sending party can define certain security requirements, such as confidentiality, anonymity, non-repudiation of origin, non-repudiation of delivery.

An **exchange** consists of several combined transfers: two or more parties have the *assurance* that if they transfer something specific to the others, they will also receive something specific.[2] Note that we consider a transfer with non-repudiation of submission or delivery as a "transfer", and not as an "exchange."

Begin and end of a transfer or exchange will be clear for each party locally, but there is not necessarily global synchronisation, i.e., different parties might recognise the same logical events at different times or in different orders.

The actual sequence of transfers and exchanges in a business session can either be determined directly by the user, or it can be described by a protocol for such business sessions. Several exchanges and transfers may then be linked via a common context. Of course, a protocol may branch, i.e., allow more than one sequence.

After each transfer or exchange, the parties are either

- **satisfied** and thus willing to proceed with a certain number of other transfers or exchanges or
- **dissatisfied**, in which case an *exception* or *dispute handler* is raised (which might end up at a real court if all else fails).

[2] We require a *guarantee* of fairness for exchanges. If no such guarantee is required, we model such a conversation by several transfers.

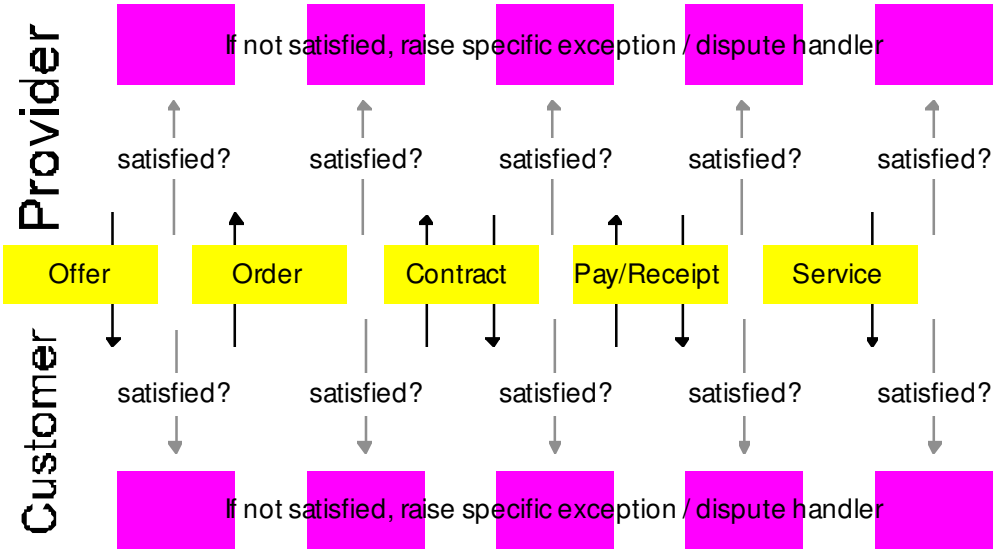


Figure 9 Simple example of a sequence of exchanges and transfers

The protocol might enable other sequences as well, e.g., after "Contract" "Payment without Receipt" might also be enabled. Note that some business sessions require exchanges and transfers between more than two parties.

A **container** is a general data structure for what can be transferred and exchanged. It may contain several **primitive types**:

- signed documents, such as
  - certificates;
  - receipts;
  - signed statements;
- information, such as
  - digital goods;
  - information necessary to access a service (e.g., an address and password, or a cryptographic key that protects a video stream);
  - information necessary to access physical goods;
- electronic money.

Information means something that is not interpreted for the purpose of a specific transfer or exchange, i.e., it is just transferred as a bit-string. (Of course, it *is* interpreted in the protocol for the business session, e.g., as good or bad digital goods.) In contrast, the transfer of a signed document primarily means that the recipient must get a correct signature. Money, as it occurs in the business session, is primarily a description, such as “350 ECU via Eurocard”, and the implementation of a transfer of money will typically consist of several internal transfers of payment details.

Figure 10 gives an overview of the possible exchanges of primitive types. Transfers are included as exchanges of “something” for “nothing”.

Party A sends → Party B sends ↓	Money	Signed document	Information
nothing	Payment	Certificate transfer etc.	Information transfer
Money	Fair money exchange	Fair payment with receipt	Fair purchase
Signed document		Fair Contract Signing	Fair conditional access
Information			Fair information exchange

Figure 10 Transfers (first row) and exchanges of primitive types

We expect that the listed transfers and exchanges are sufficient in order to build all relevant commerce services. Obviously, the matrix of Figure 10 is complete with respect to pairs, but there may be different security requirements in detail.

Our initial investigation is two-party centred. The same considerations can be applied to the multi-party case. For instance, more than two parties might wish to sign a joint contract, or one sender might want to send a certified mail to several recipients.

An important goal of later sections is to define a toolbox (e.g., as a set of generic services, or as a configuration / script language) for transfers and exchanges, so that business sessions can easily be composed out of a few generic building blocks.

4.3 Exception and Dispute Handling

All services have to define what happens if one of the parties is dissatisfied at a certain stage, i.e., how an **exception** is handled. Exceptions may be raised because of

- a party having insufficient privileges,
- an unintentional error, e.g., a lost message, incompatible software versions, or a misstroke by the user,
- malicious behaviour by others, e.g., abuse of the credit card number of someone else.

Obviously, there is no well-defined difference between the second and the third possibility. In particular, the reason for the exception is not known at first, e.g., when the dissatisfied party notices an incorrect transaction on its credit card statement. Therefore one starts with **exception handling** in the narrow sense, which is based on the assumption that all parties are honest. If this optimistic approach fails, the parties are in a real **dispute,** and more pessimistic approaches must be used, usually involving **arbiters** (and finally courts).

Exception handling requires all parties to keep sufficient **audit trails** of what has happened so far, so that they can try to agree on the current state. For a real dispute, the audit trail has to contain evidence (e.g., non-repudiation tokens) that can be shown to and verified by the arbiter using a **decision procedure**.

For legal certainty, the result of a decision procedure must be well-defined and predictable. But there may be protocols where the result is not necessarily “correct” in the sense that it restores a previous state where all honest parties are satisfied. For instance, in current mail-order/telephone-order transactions using a credit card, disputes between buyers and sellers about an order cannot be resolved unambiguously. In these cases, additional rules define how to resolve such ambiguous situations, e.g., they might say that the buyer is never liable for such orders.

Other disputes (e.g., due to misstrokes or incompatible software) may not be possible to solve until a case has been in court and a judge has interpreted the law.

Exception handling might require to switch to another communication protocol, e.g., from http to asynchronous electronic mail, but should remain within the digital system as far as possible. Actions outside, like phone calls or sending letters with hand-written signatures, should be minimised, but may remain necessary as a last resort.

5 Service Architecture Overview

The *SEMPER* architecture consists of four *layers*, see Figure 11.

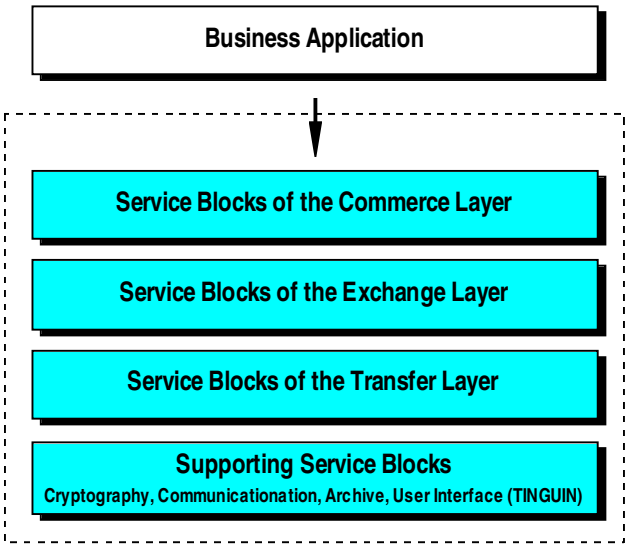


Figure 11 Architecture of *SEMPER* — Overview

5.1 The Business Application

The Business Application Layer is the environment where the *SEMPER* library is used. Currently, http servers are implemented to embed the *SEMPER* library into the World-Wide-Web (see Chapter 12 or deliverable D04 for details).

A business application is implemented using the services of the lower layers, primarily the commerce layer. Access control is used to prevent unauthorised access to services of the layers below the commerce layer. Note that this may allow certain applications to access services of layers below the commerce layer. An example where this is useful is when a user is being registered (this is a business application in *SEMPER* and is most easily handled by letting the application access the certificate service block directly).

5.2 The Commerce Layer

The Commerce Layer provides services that directly implement protocols of business sessions, e.g., how specific merchants or types of merchants handle customer registration and offering, ordering, payment, and delivery of goods. Later, we will provide a secure downloading mechanism to enable clients to interact with formerly unknown server commerce services and still retain security. Naturally, this involves some kind of code certification.

However, the commerce layer not only offers entire such protocols, but also building blocks that may be of more general use, in particular for buyers. In particular, these standardised services may offer template services, e.g., to display and fill-out standardised order-forms and construct containers from them.

Currently, only these basic building blocks are implemented whereas the control flow is done by the business application and downloading of services is not implemented.

5.3 The Exchange Layer

The Exchange Layer will provide services for the fair exchange of containers. “Fair” means that, if A and B agreed on what to exchange, Party *A* receives *B*’s container if and only if *B* receives *A*’s container. (Without fairness, this would be just two transfers.) It will be implemented in year 2 of SEMPER.

5.4 The Transfer Layer

The Transfer Layer provides send, receive, and composition services for containers in order to transfer information between client and server according to given security attributes.

As mentioned in Section 4.2, everything that can be transferred by the transfer layer is encapsulated in one data type called **container**. Each container can contain several basic items, such as signed documents, information, and money. Moreover, the transfer or the container may have **security attributes**. For instance, the sender (more precisely: the invoking entity) specifies that the transfer of the container should be with non-repudiation of origin; then the sender’s entities of the transfer layer, with the help of the supporting services, attach a suitable signature and send everything off, and the recipient’s entities of the transfer layer verify the signature and tell the recipient something like “you received this container, and the sender cannot deny the transfer”.

This layer contains the **transfer manager**, the **payment service block**, the **statement service block**, and the **certificate service block**. Basically, methods of the transfer manager decompose containers to be transferred, and let signed documents and information be handled and transferred by the statement service block and payments by the payment service block. The peer transfer manager reassembles the received parts of a container to the original tree-structure. The certificate block helps in the administration of the necessary certificates and performs user-registration procedures.

In addition to the actual transfers, the transfer layer also offers invoking entities to manage these transfer services, e.g., to manage the contents of an electronic purse or to set up relations to enabling third parties. In contrast to the actual transfers, such services do not need to go via the transfer manager, e.g., the payment service block can immediately be used by higher layers for purse management.

5.5 Supporting Services

The Supporting Services provide support to all the upper layers.

The **Cryptographic Services** provide the cryptographic services needed for data security. Examples for such services are message encryption and decryption, hash functions, message authentication codes, digital signatures, and their key generation. The crypto block is entirely used as a supporting service and does not provide a transaction class allowing the creation of crypto sessions.

The **Communication Services** support communication between different entities. The services offered are independent of the network and protocols actually used, i.e., they

are defined only via quality of service parameters. Thus the communication service block shields the other upper layers from network details.

The **Archive Services** provide local storage and archiving of all persistent data. Examples are digitally signed messages, certificates, cryptographic keys, and transaction or evidence objects. The archive includes features for secure storage of data by, e.g., encrypting the data or archiving in tamper-resistant memory (e.g., on a smart card).

The **Trusted Interactive Graphical User Interface Services** support the interaction between services and the human user. The basic idea as that by providing a distinguished user interface to the security services, the user is aware that all inputs through this user interface are critical and that the "usual" business application user-interface (e.g., www-browser) must not be used for any critical input.

Naturally, the security services user-interface must be under control of the user's own device. For instance, if a buyer verifies the identity of a seller, the message "the verification of the identity was correct" cannot come with the seller's html-pages dependably: Any crook could simply provide this message ready on its html-page! Instead, the user's own device must show this message in a way so that the user can recognise it as a message from her own device. Ideally, the TINGUIN would reside on a mobile user interface, such as a PDA [PPSW 95].

Moreover, ergonomics has to be considered in the design of the TINGUIN even more carefully than usual, because any misunderstanding by the user may destroy security.

The **Preference Services** provide a uniform way of handling preferences for each service block. Each manager must define the available options and the user selects the preferred ones through the TINGUIN. In addition, the preference block stores the actually installed configuration, i.e., all information specific to a particular installation like the target system, the memory size, the screen type or any other hardware devices currently supported.

Access Control Services. provide all services needed to implement access control on the methods of the various blocks of the *SEMPER* architecture. Access control ensures that potentially dangerous, compromising or undesirable operations cannot be performed by unauthorised users or without the user's explicit consent. Access control is *capability* based. Each block must declare a capability for each action for which unconditional access is undesirable. Such a capability must be verified each time the corresponding method is called. The Access Control Block provides the necessary services to manage such capabilities based on roles. By the way, we assume that they only try to access each other nicely by their services, i.e., that the operating system does not allow them to tamper with each other's code or memory.
The roles are currently unlocked after appropriate user authorisation. Later, this should be based on a policy fixing

- how and at what times the user device makes sure that the correct user is there, *identification* (also called *user access control*),
- and at what times the user must be asked for an OK, *authorisation*, in particular whether higher layers can ask for authorisation for many actions, e.g., many small payments, in advance, that would have to be authorised by the user if they were carried out individually.

The service blocks described so far are shown in Figure 12.

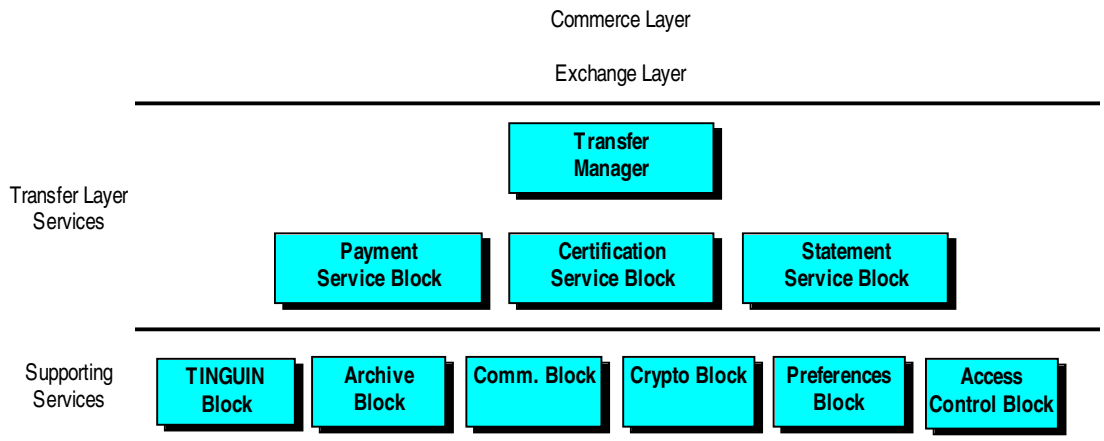


Figure 12 Service blocks of Supporting and Transfer Services

In addition, the actual implementation also contains a **utility block** with various utility services for, e.g., converting objects into streams of bytes, and for logging and debugging.

The whole architecture has to be considered specifically to provide for **trusted hardware** on small mobile devices (ranging from a smart card that can *only* be used together with another device of the same user, to a PDA that is used on its own). Of course, if trusted hardware is available at all, as much as possible should be implemented on it, but there are limits in program storage, computing capability, and the quality of the user interface.

6 Set-up, Registration and Certification

This section describes the necessary entities and functions needed to get started with *SEMPER* and to solve disputes.

6.1 Infrastructure

Before becoming a player on the marketplace (mostly a user, i.e., a buyer or seller), one has to obtain the necessary software and hardware and access to an underlying network. As this affects the user's security, it must be done carefully:

1. A user must be able to trust in the *design* of the components, i.e., in the claims made about the security and functionality of these components.
2. Users must be able to trust that the *actual components* received are implemented according to the trusted design; i.e., they have to trust in the specific code copied from an ftp archive or received on a disk or CD-ROM, or in the specific smart card received from a bank.
3. Users must be able to trust that the *machine* and code they are actually using are not unintentionally or maliciously manipulated during operation, e.g., by an error in the operating system, a computer virus, or a Trojan horse.

Solving these problems in general is almost impossible and certainly out of scope of SEMPER. The approach used for the *SEMPER* prototypes and trials is as follows.

1. The design of *SEMPER* itself will be open and public, which will allow security evaluations. Clearly, this may not be possible for the typical user. However, it allows independent verification of the components of *SEMPER*. It will be clearly indicated which existing basic components (like payment systems, cryptographic tools) have been used to provide the actual security services, but the *SEMPER* consortium will not take actions to increase trust in components not developed by the consortium itself.
2. The software required for participation in the initial *SEMPER* trials will be available at ftp archives and additionally distributed via disks. The *SEMPER* software will be digitally signed using generally available security tools (e.g., PGP) by a designated participant of *SEMPER* (other partners can verify this signature, thus the correctness of the signed package will essentially be the responsibility of the consortium, not a single partner). The fingerprint of the public key used by the *SEMPER* consortium will be published reliably, e.g., in journal publications of *SEMPER* and on public http servers. A user getting the software can now verify the signature against the (fingerprint of the) public key. To increase credibility, a recipient can retrieve and compare copies of the *SEMPER* software and the public key from different sources.
3. *SEMPER* supports the use of various hardware platforms including tamper-resistant hardware. However, it does not enforce tamper-resistance, and in the first year trials such hardware will not be available. This is important for the openness of the architecture for future developments, in particular in the area of mobile devices. It is currently the user's responsibility to protect his or her machine as welle as *SEMPER* software against malicious modifications. Note, however, that by verifying the

SEMPER software against the signature received originally, changes to the components can be detected.

6.2 Certificates

We define a **certificate** very generally to be any signed statement about a person and distinguish three types of certificates:

- **Key certificates**: Here, the statement is that a particular public key belongs to a particular identity (distinguished name). For instance, the well-known X.509 certificates are of this type.
- **Attribute certificates**: These certify attributes of either the person (identified by a distinguished name) or of a certain key (identified by a certificate identifier) of this person, e.g., that the person is willing to accept legal responsibility for signatures made with this public key under certain types of statements. These certificates may also be used to give proxy to a user.
- **Hybrid certificates**: These contain both a public key and attributes, typically a description of what this public key can be used for. They are mainly intended for service providers issuing certificates particularly for their own service.

In the cryptographic literature, the term *credential* is used instead of hybrid certificates, but we use this term for the personal information shown by the user at registration.

The exact data in a certificate, e.g., expiration dates, algorithm identifiers, the identity of the certifying authority, and how much **responsibility** the certifying authority is willing to assume about the signed statement (e.g., depending on how well the user's identity was verified in registration), will depend on the actual registration and certification module. See the description of the certificate manager for these details.

6.2.1 Meaning of a Certificate

A certificate is a signed statement, and thus, if the recipient acts on it and later finds out that it was wrong, he can hold the signer responsible for it, as far as the responsibility indicated in the certificate goes. We now emphasise some particular points that sometimes cause confusion.

A certificate does not mean that the statement certified is true, only that the certification authority says so. It is in the recipient's judgement to believe this authority or not. In some cases, the fact that the certification authority assumes responsibility is enough, but in some cases it would not help the recipient much if the certification authority is punished or has to pay damages after the fact.

Someone signing a certificate only assumes responsibility for the statement signed. In particular, someone signing a key certificate is only responsible for correct linkage between person and key, and in no way for this person's trustworthiness. Similarly, someone signing an attribute certificate that the person has declared herself *willing* to accept legal responsibility for certain types of statements is not signing that this person is *able* to do so — this is a typical attribute of the key, in contrast to an attribute of the person, such as a credit limit. A physical analogue is a passport: It links a certain hand-written signature to a certain identity, and confirms (given laws about hand-written signatures) that this person will be held responsible for any document signed with this

signature, e.g., cheques over high values. But of course the passport issuer does not in any way suggest that this person has an arbitrary amount of money to back these cheques, or even assume responsibility for these cheques.

We deal with the semantics of certificates, not with their internal data structure. In an implementation, someone who only signs one type of hybrid certificates, i.e., always with the same attributes, may not include the attributes explicitly. In particular, this may hold for some legacy certificates. However, for an open architecture like *SEMPER,* there must be a standardised method for recipients of such certificates to find out what the certificate means.

A certificate in itself does not make the person responsible. Only a person's own declaration of intent towards the registration authority will imply this. If the authority is afraid that *A* might later deny this intent, he has to store evidence, such as a hand-written signature by *A*.

6.3 User Registration and Certification

User registration and certification is required for some security services, e.g., subscription to an electronic payment system or registration of cryptographic keys needed for digital signatures. This registration is necessary in order to link electronic actions to physical organisations and individuals, which is the basis of all kind of commerce.

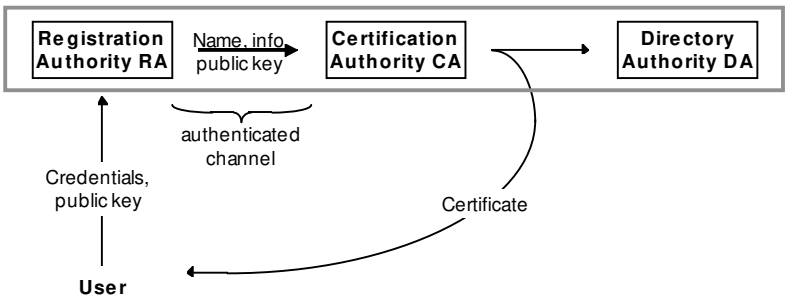


Figure 13 Authorities and flow of information

1. Registration and certification is usually described as a two-step procedure. The user registers his public key and credentials (e.g., name, address, bank account) at a registration authority RA). The registration authority verifies all information supplied by the user (their credentials). If the information is correct, the registration authority asks the certification authority (CA) to issue a certificate corresponding to the registered information.
2. The certification authority issues a certificate and sends it to the user. Furthermore, the certification authority sends the certificate and the registered information to the directory authority (DA).

In the following, these three types of authorities are discussed in more detail. Note that quite different parties may play the roles of such authorities, depending on the type of certificate to be issued. The distribution of responsibility between these authorities are discussed in Section 6.3.4.

6.3.1 Registration Authority

A registration authority registers and verifies all information supplied by the user when registering. In particular, at least for a user's first key certificate, it is responsible for defining the suitable distinguished name (unique identifier) for the user. Note that the verification of the information supplied by the user is not trivial: For initial registration it may even include that the user visits the RA for signing a paper contract by hand.

The communication between a registration authority and the corresponding certification authority must be secured adequately (at least authenticity is required).

6.3.2 Certification Authority

A certification authority has four responsibilities:

1. Given registration information from a corresponding registration authority, it *issues* a certificate. The certification authority trusts that the registration authority supplies correct information. The certificate is sent to the user, and the certificate and the registered information are sent to one or more directory authorities.
2. *Extending* a certificate. This simply means that the validity period is extended. It can be done given a (possibly electronic) request from the user.
3. If a user requests a certificate to be *renewed*, he will have to generate new keys and ask the certification authority to certify the new key pair. As long as the user has a valid certificate he does not have to register at a registration authority in order to renew the certificates. If a certificate is renewed, the old one may, depending on the policy of the CA, be revoked as described below.
4. The certification authority may revoke certificates if it is suspected that the key is compromised or was registered by an impostor in the first place, or in the case of certain attribute certificates, the user is abusing the granted rights. Revocation is either initiated on request from the user (possibly electronic) or by the certification authority, e.g., once compromise is suspected. Note that in case of compromise anybody knowing the corresponding secret key will be able to make an electronic revocation request on behalf of the owner. A certificate is revoked by issuing a revocation certificate, which is similar to normal certificates, except that it indicates that the certificate is revoked, and the reason for revocation is given (e.g., suspicion of compromise).

Whenever a certification authority issues or changes a certificate (extension, renewal or revocation) it will communicate the new information to the directory authority for updating.

There may be certificates for which the certification authority assumes a different degree of responsibility depending on how well the credentials of a user have been verified. This level must be clear from the certificate. For instance, a user might have to show a passport and to sign a contract with the registration authority, and might receive a certificate on a public signature key and a smart card, and also a signed contract from the certification authority. Or the user might have to send just their address to the registration authority, and might get a password.

6.3.3 The Directory Authority

A directory authority maintains a database of certificates, valid or revoked, and other registered information received from a certification authority. The directory authority is responsible for answering queries to the database and for sending out lists of revoked certificates ("black-lists") at regular intervals if this is required.

The directory authority is trusted to store the information reliably, especially revocation certificates. Note, however, that a directory authority cannot forge certificates, because only certification authorities have the corresponding secret keys, and thus need not be trusted in this sense.

Preferably, the communication between the directory authority and the user is secured (authenticity).

6.3.4 Trust Among the Three Types of Authorities

Initially, the responsibility for an existing certificate naturally lies with the certification authority which has signed it. However, if a certification authority is run by a different legal entity than the registration authority an agreement will distribute the responsibility among them.

Implementing the two-step procedure for registration and certification using physically and organisationally separated authorities has the disadvantage that more trust between registration authority and certification authority is introduced. The main advantage of separated authorities is that it allows to combine central certification with local registration, thereby allowing the user to show up in person at the registration authority easily.

A physical separation of directories from certification is certainly useful even if they are run by the same legal entity, because a directory, which answers on-line requests, is naturally quite vulnerable to attacks on the computers, whereas the certification process as such should be run in a much more secured environment on a special-purpose computer. Furthermore, it is useful to store certificates in many places for fast access.

Revocation certificates make the trust relations complicated. A revocation runs through several steps, e.g., from the user initiating it over certification authority and directory authority to the recipient. If a recipient acts on a certificate that was revoked, responsibility can only be assigned if it is clear at what time the recipient did so, and how much time each entity in the chain is allowed before it becomes aware of the revocation and passes it on. In particular, this makes it complicated to replicate directories.

6.3.5 Certificates in *SEMPER*

The *SEMPER* architecture provides services for registration and certification and the use of the certificates, but neither prescribes who can play the role of which authority, nor for what purposes prior registration or certification are required. A certificate manager in provides an interface to actual modules implementing the services needed for registration, certification, and directory authorities.

Obviously, there are many possibilities as to what kinds of certificates are required for certain actions on the marketplace. Assume that a seller *S* and a buyer *B* interact, some important examples are:

1. *S* and *B* does not require registration from each other.
2. If *S* requires *B* to pay before getting goods, *S* does not require any registration or certificates from *B*. (In the background, however, *B* will have registered with a payment system provider.) However, *B* would be wise to require certificates from *S*, at least about the claimed identity of *S*, and possibly about its trustworthiness and the reputation of its goods. This is typical for on-line purchase of information.
3. If *S* delivers first, or takes other trouble before *B* pays (e.g., in air ticket reservation), *S* might want to verify that it will eventually get paid, and thus that *B* has already registered with a third party, *T*, (so that *S* can at least verify that the order truly comes from *B*), and possibly *B* has some certificates.
4. For specific services, *S* might require *B* to register with *S*. This registration might depend on certificates *B* already has.

Case 1 is certainly the most attractive one, since it allows everybody to interact freely. But it can be applied only if either nothing legally meaningful happens at all, e.g., if *B* only looks at *S*'s catalogue, or if security is ensured out band, e.g., if an order must be printed, signed, and sent via mail before fulfilled.

Cases 2 and 3 are typical relations between sellers and buyers, as explained above.

Case 4 is how, e.g., payment systems work, i.e., if *S* is not a seller, but a payment system provider. The actual registration might be done by *S* itself, or on behalf of *S* by a third party. Registration with *S* might depend on credentials which are based on other registrations. For instance, one can open a bank account provided one can show a valid passport, i.e., registration with a bank is actually based on registration with a government.

The architecture of *SEMPER* supports all these cases. In particular, third parties in which any form of trust is needed should be registered and certified more strictly.

7 Architecture of the Exchange Layer

The exchange layer provides fair exchange services which implement the atomic actions of our model of electronic commerce (see Section 4). Fairness in this context means that either the complete exchanges takes place, i.e., both participants receive what they expect, or nothing happens at all. Examples are exchanging a payment for a computer program or exchanging signatures under a contract. In this chapter we first describe the data structure for business documents, then we sketch the services to be provided by the exchange layer.

7.1 Containers

The business documents transferred by the exchange and transfer layers are called **containers.** A container is a tree structure in which the leaves contain

- signed documents,
- information, or
- electronic money.

Nodes containing information will not be interpreted by the exchange layer. Thus only statement and payment nodes are of interest at this layer.

In addition to containers containing goods we also provide containers describing what a participant expects, i.e., containers which describe goods and security attributes.

7.1.1 Containers with Electronic Goods

The internal nodes of a container specify security attributes together with their scope, see Figure 14.

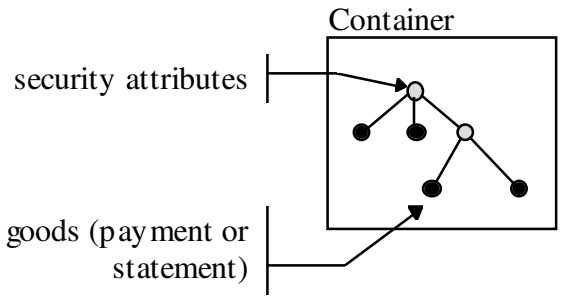


Figure 14: Container with Goods (Grey nodes depict security attributes, black nodes goods).

The security attributes currently supported by the transfer manager are described later. Examples are

- Confidentiality
- Anonymity and unlinkability
- Integrity
- Message authentication
- Non-repudiation of origin/submission/delivery/receipt.

The scope of the security attributes describes for which leafs this security attribute is valid. The scope may be specified by restricting the validity of an attribute to the left and/or right sub-tree. In addition one may also limit the depth or the protocols to be used by, e.g., specifying that all children should be sent together.

7.1.2 Containers Specifying Expectations

In addition to goods, fairness of exchanges requires some means to specify the container expected in exchange for the container sent, i.e., in addition to containers containing electronic goods and security attributes, we also provide containers describing goods and security attributes. Examples for combinations of containers describing an expectation with containers containing goods are

- I expect 25$ in return for the data (program) I'll send.
- I expect a signature under the fixed text of a contract in return for my signature under it.

Specifications of containers have the same structure as containers: They contain descriptions of goods in their leafs and descriptions of security attributes in their internal nodes.

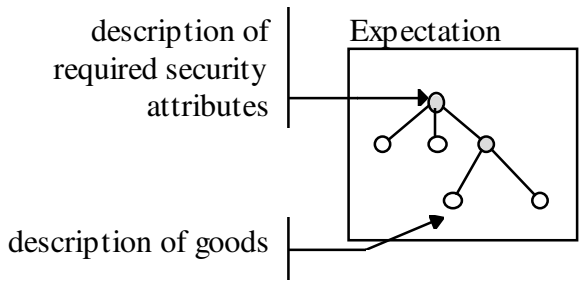


Figure 15: Container with Goods (Grey nodes depict descriptions of security attributes, white nodes descriptions of goods).

One major problem of fair exchange of containers are appropriate mechanisms for expressing the expectations of both parties which may be checked automatically. For some kinds of items, such as payments, this can be done straightforward by specifying the attributes of the payment. However, for most purchases of data or programs it will not be possible for the exchange manager to check this.

In order to handle this problem we separate syntactical correctness and semantic correctness:

Syntactic Correctness: Did the parties agree on the description of the delivered goods, e.g., did they both agree that the item described "on-line-course about digital signatures" was supposed to be delivered.

Semantic Correctness: Does the data contain the agreed goods.

The first problem can easily be solved as an exchange should only occur, if the descriptions of the items to be delivered match those of the items expected. The second problem will in most cases need human interaction by an arbiter. However, our approach simplifies the work of this arbiter considerably: As input the arbiter gets the description on which both parties committed together with the data. The only decision he has to make is if the data matches this description.

7.2 The Exchange Layer

The exchange layer provides services for fair exchange of containers. In the following we only describe two-party exchanges but this can easily be extended to the multi-party case. An exchange involves two parties *A* and *B*. Each party inputs a container. An exchange without fairness requirements outputs *A*'s container to *B* and *B*'s container to *A*. This can be done by two subsequent transfers.

Fair exchange, however, additionally involves

atomicity: *A* receives *B*'s container if and only if *B* receives *A*'s container, or

fairness: an exchange only takes place if *A's* container matches *B's* expectations and *B's* container matches *A's* expectations.

For fairness, we assume that *A* and *B* beforehand has agreed which items they want to exchange. Each party inputs what they expect to receive and what they would like to send. A major problem is that for many expectations it is impossible to check automatically that they are fulfilled, e.g., how could a machine check that a delivered on-line course is really about digital signatures.

7.2.1 Model and Requirements for Fair Exchange

We now describe the requirements for the two-party exchange of containers. We assume that A and B each have a container and want to exchange them according a previous agreement, i.e., the two items to be exchanged are characterised in advance, typically by a 2-party contract of *A* and *B*.

Just requiring that the containers input by A and B are mutually exchanged is not sufficient. They may input arbitrary containers, i.e., they may cheat each other by inputting containers which do not match their previous agreement for the exchange. The general setting is depicted in Figure 16 and Figure 17: A and B input their containers, with goods and with the specification what they expect to receive (their local view of the agreement) together with a description of the security attributes of the exchange. An exchange should only occur, if the description of the expected items matches the offered items and if the security attributes required by both parties do not contradict each other.

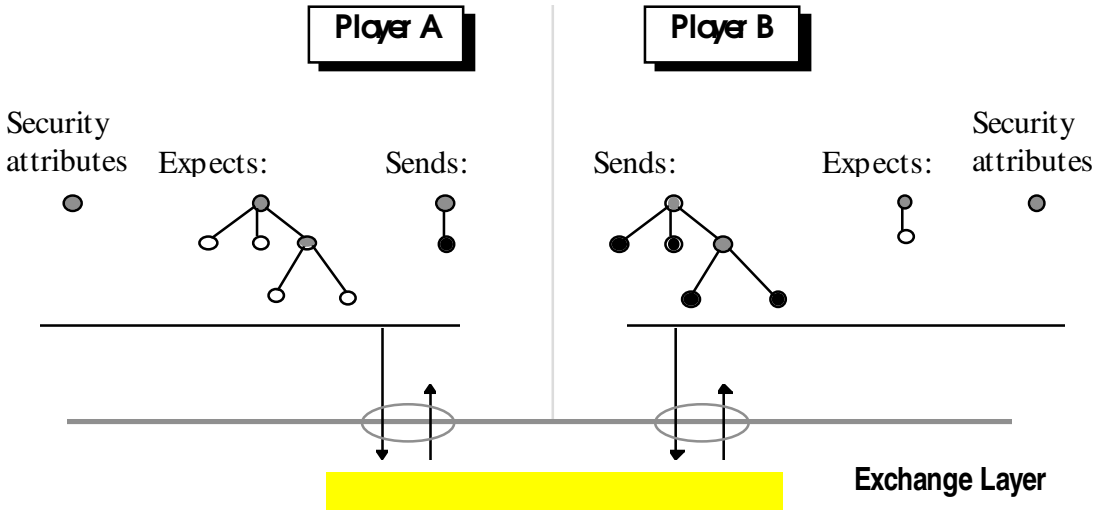


Figure 16: Example input for fair exchange: Both parties input an item, a description what they expect to receive, and the security attribute of the exchange.

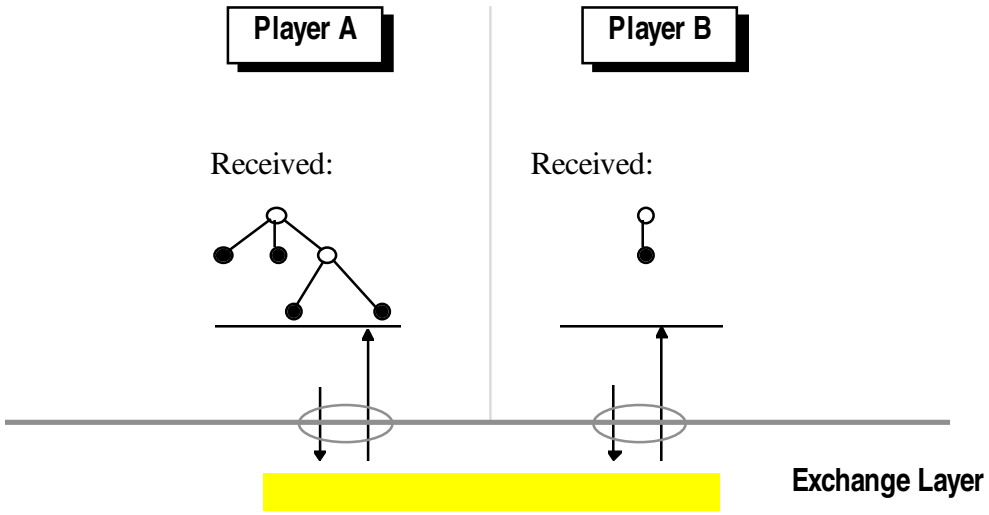


Figure 17: Example output of fair exchange.

Security and Functional Requirements:

1. *"Fairness for A:"* If honest *A* (and if an involved third party *TP* is honest, too) receives output "failed" then *B* has *not* received $item_A$ (i.e., *B* has not received any additional information about $item_A$). This holds even if *B* is dishonest. "Fairness for *B*" is defined symmetrically.

   Some protocols might satisfy only a weaker fairness requirement:

   1b. *"Weak fairness for A:"* If honest *A* (and if an involved third party *TP* is honest, too) receives output "failed" and *B* has received $item_A$ then *A* can prove to an honest third party that *B* has received $item_A$ (and *A* has not received $item_B$).

2. *"Correctness for A:"* If honest *A* receives some item $x \neq$ "failed" then the description contained in *x* must satisfy $spec_B$ and *x* must satisfy its description.

3. *"Correctness of dispute for A:"* If honest *A* (and if an involved third party *TP* is honest, too) has received output "failed" then no honest third party will decide against *A* in [1b].
4. *"Confidentiality for A and B:"* No third party does receive any information about the items exchanged, nor about their specifications.
5. *"Anonymity / unlinkability for A:" B* does not receive information about the identity of *A* (cannot link this exchange with any other transaction).
6. *"Availability for A and B:"* If *A* and *B* are honest than both parties will receive the required items.
7. *"Disputability of Third Party Services:"* If *A* and *B* are honest and the network works dependably but the protocol does not provide the desired result, then *A* and *B* can prove that a third party has not worked properly.

Third parties: Usually fair exchange is implemented using one or more third parties. In this case the third parties must be specified in advance (e.g., as part of the contract that describes $spec_A$ and $spec_B$), and for most security requirements at least one (or "all," or "a majority of all," etc.) must be assumed to be honest. The required trust in third parties is an additional parameter for the security attributes of the exchange.
Disputes: Necessarily requirements on the result of a dispute presuppose that the third party that decides on the result of the dispute is honest.

7.2.2 Exchange Protocols

We describe the services of Exchange Layer informally and for the 2-party case only. Most of the services have natural extensions to the multi-party case, e.g., multilateral contract signing. Note that for some services a new requirement must be considered: consistency. For instance, secure communication of many parties would include secure broadcast where all recipients want to be sure that they receive the *same* message, even if the sender or some of the other recipients are dishonest (in the area of distributed algorithms this is called the Byzantine Agreement problem).

The protocols which are currently designed are based on an optimistic approach, i.e., after a mutual agreement between the participants fixing the goods and a third party for recovery, both parties send their containers [AsSW 96]. If a fault occurs, the parties may complain at a third party which will restore fairness depending on the items exchanged by, e.g., undoing a payment or creating affidavits. Only if such an optimistic approach is not possible, the third party will be actively involved in the exchange protocols.

8 The Commerce Layer

The *Commerce Layer* provides the instantiation methods for connections between business applications on clients and servers including methods for commercial business applications. The basic classes for doing this with are the *CLConnection* class and subclasses derived from the abstract class *CLServices*.

The principal modus operandi can be described in brief (according to the *SEMPER* architecture): A customer starts his computer with the SEMPERClient software. He then invokes a web browser thus loading a page from a seller which uses the SEMPERServer software. By selecting a service offer from that page via a 'localhost'-link already furnished with appropriate parameters the SEMPERClient task is invoked thus being passed those parameters (see Chapter 12 for more information on the integration of business applications).

The SEMPERClient now parses the parameters and uses these values to establish a connection between the customer's (client) computer and the seller's (server) host by instantiating a *CLConnection* object. The SEMPERServer on the seller's host is waiting for a connection request and instantiates the corresponding *CLConnection* object on server side.

Client and Server will then call the appropriate business application on their sites passing them the *CLConnection* objects representing the connection that has been established.

The client business application is now to perform its desired transaction by creating one or more instances of different CLServices subclasses such as CLOfferRequest and invokes the provided send method(s), i.e., in this example to get an Offer for certain goods/services provided by the seller's server.

During the instantiation of the *CLConnection* object a so called TINGUIN display was created, which allows important and security relevant information to be shown via this trusted interface.

In the following sections the basic classes and services will be described in more detail.

8.1 The CLConnection Class

A pair of instances of this class represents an established connection between a client and a server that can be used by the corresponding business application on either site. There are two different constructors to be invoked:

CLConnection(<parameters>)
This constructor is to be called by the client site. The parameters supplied contain all the information necessary to establish a correct connection to server's site, e.g. server address and name of the business application.

1. A new *GroupTransfer* object will be instantiated to provide send and receive methods to deal with the underlying Transfer Manager.
2. A TINGUIN display session is initialised and is used during this application.

3. An INIT message containing initialisation data is sent to the server and an ACK message is expected in return. Once the acknowledgement has been received the constructor returns indicating that a new connection has been established. This connection object will be passed to the client business application started by the SEMPERClient software.

CLConnection()
Constructor to be called by the server. All necessary parameters will be supplied by a client's request for connection.

- A new *GroupTransfer* object will be instantiated to provide send and receive methods to deal with the underlying Transfer Layer.
- Once a connection request from a client has been received - i.e. an INIT message containing all initialisation data including the name of the business application to be invoked on the server's site - this constructor will return an ACK message to the client.
- If the ACK message was received by the client (i.e. the send operation did not raise an exception) the constructor will return itself thus indicating a connection has just been established. Now the SEMPERServer can continue by looking up the name of the business application to be started and invoke it, passing the newly instantiated *CLConnection* object (the connection itself).

The diagram in Figure 18shows in brief the main steps performed until a connection is established:

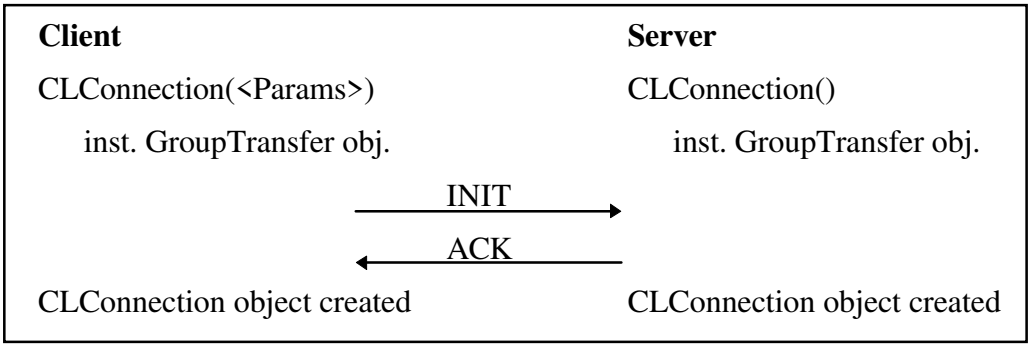


Figure 18: Establishing a connection in the commerce layer

Now that a connection has been established there are several commercial transactions that can be applied. These are described in the following section.

8.2 The Commerce Services

The basic class for every service provided by the *Commerce Layer* is the abstract class *CLServices.* It provides a send and receive method for the data exchange between client business applications and server business applications.

Before any service can be used an object of its class has to be instantiated. After instantiation the send and receive methods may be used by passing them the *CLConnection* described above.

For the moment only the services needed in the first trial are implemented. In the following the classes supporting these are described further (they are all implementations of the abstract class, *CLServices*).

8.2.1 CLOffer

- A server sends an offer previously requested by an *CLOfferRequest* of a client via *CLOffer.send*.
- A client receives the offer using *CLOffer.receive*.

8.2.2 CLOfferRequest

- A client requests an offer from a server via *CLOfferRequest.send*.
- A server receives a request for an offer via *CLOfferRequest.receive*.

8.2.3 CLOrder

- A client sends an order to a server via *CLOrder.send*.
- A server receives an order from a client via *CLOrder.receive*.

8.2.4 CLOrderConfirm

Works similar to *CLOrder* - here the user has to confirm the transaction through the TINGUIN.

8.2.5 CLOrderPay

- A client sends an order including the payment for it to a server via *CLOrderPay.send*.
- A server receives an order and a payment from a client via *CLOrderPay.receive*.

8.2.6 CLOrderPayConfirm

This service works similar to *CLOrderPay* as the client has to confirm the transaction using TINGUIN.

8.2.7 CLGeneral

This service provides a general receive for all possible commerce services mentioned above. It may be used in case a business application can receive different requests and, therefore, is not able to predict the one to be received at a time.

9 The Transfer Layer

The *Transfer Layer* provides send, receive and container composition services for the *Commerce Layer* service classes in order to exchange information between client and server. The basic classes to perform these operations are the *GroupTransfer* and *Container* classes.

When a connection is being established via *CLConnection* objects on either side (see Chapter 8) both server and client will create an associated *GroupTransfer* object which will be used as the underlying means of data transportation. The *GroupTransfer* class itself according to the *SEMPER* architecture uses the *Communication Block*'s services to build a connection and to deliver data to its destination.

Each *Commerce Layer* service class derived from *CLServices* can through the *CLConnection* object, which it gets on invocation, use send/receive services of *GroupTransfer* to transmit containers (i.e., data tree structures created with the methods of the *Container* and *Node* classes).

Before a *GroupTransfer* object sends the data to its peer entity the tree nodes are interpreted and the appropriate methods of the *Statement Block* and the *Payment Block* are invoked.

The data will be delivered via the corresponding *CLConnection* object to the receiving *Commerce Layer* service which will now extract the information needed and pass it to its business application.

9.1 The Transfer Layer Services

The following services are provided to the *Commerce Layer* once a connection has been established and the *GroupTransfer* objects have been created:

- services to send and receive containers according to the requested security requirements via the *GroupTransfer* class.
- services to compose a container data structure using *Container* and *Node* classes.

9.2 Basic Classes

The *Transfer Manager* mainly deals with the following essential object classes, which will be described in the following sections: *GroupTransfer*, *Container*, *Node*, *PortManager*, *Security* and *TransferData*.

9.2.1 GroupTransfer

Provides the send/receive service for the *Commerce Layer*. Each *CLConnection* object creates its own *GroupTransfer* object to establish a connection between client and server. This object uses the *Communication Block*'s facilities (see Section 11.1) to finally build up the connection. An exception will be raised if there is no way to instantiate a connection.

Once a *GroupTransfer* object exists on both sides of a connection, *Commerce Layer* services can use it to exchange transaction data tree structures stored in containers.

Every time a send method is invoked it first processes the container by calling the appropriate services according to the node type, e.g. methods of the *Statement Manager* in case of statement nodes, etc.

In brief, the main methods for data transfer between a Client and a Server are:

GroupTransfer()
Class constructor that establishes a connection between a Client and a Server using the services of the *Communication Layer*.

receiveContainer()
Method to receive a data tree structure stored within a *Container* structure. Optionally, a time-out value may be specified, in order to infinite waiting.

sendContainer()
Method to send a data tree structure (usually created by the upper *Commerce Layer*) to the remote side. Before the container is sent its content will be processed, i.e. the data tree contained will be traversed and every node will be processed according to its type: e.g. for statement and payment nodes the Statement Manager and Payment Manager will be invoked, respectively.

9.2.2 Container

This class that comes along with the *Node* class provides all methods necessary to compose and create containers containing the data tree structures used by the commerce services of the Commerce Layer to perform the desired transactions (e.g. a *CLOrderPay* service needs to transmit a tree structure consisting of a statement and a payment node).The main methods to deal with containers are:

Container()
Constructor method to create a container to fill it with a data tree structure.

addChild()
Adds a node to a data tree as a child node of this tree.

getRoot()
Returns the root node of a data tree stored in a container.

concatenate()
Concatenates two containers and merges their data trees.

9.2.3 Node

The node class is the brick on which container trees are build. This class provides all the necessary methods to assign values to and read private variables of *Node*. Nodes will be processed by the *GroupTransfer* object depending upon their type (e.g., statement or payment).

There are many methods that deal with the manipulation of node-internal data so just a few will be mentioned here:

Node()
Class constructor to create a Node object.

setNodeContent()
Provide data for the node.

getNodeContent()
Obtain a node's data.

setNodeType()
Set a node's type (e.g. „payment" or „statement").

getNodeType()
Determine the type of a node (e.g. „payment" or „statement").

insertChild()
Appends a node to another one as a new child.

9.2.4 PortManager

The *PortManager* class provides the management of different ports (as part of an Internet address, see Section 11.1) for different entities. All methods are static.

For each port only one *ServicePointThread* (see Section 11.1) can be opened. For this reason it is not possible to start two processes (threads) using the same ports (e.g. two clients or client and server). In order to start two or more clients there have to be threads (one for each) in one process.

There are two public methods dealing with Service Points:

openServicePoint()
Opens a new Service Point for a given port.

getServicePoint()
Request for a Service Point on a given port. If none exists, a new thread will be started and a new Service Point corresponding to the port will be created.

9.2.5 Security

This class contains all security attributes to be applied on a container and single nodes that are to be sent between client and server. The following attributes are provided:

- MESSAGE_CONFIDENTIALITY
- MESSAGE_INTEGRITY
- MESSAGE_AUTHENTICATION
- ANONYMITY_BY_ALIAS
- NON_REPUDIATION_OF_DELIVERY
- NON_REPUDIATION_OF_RECEIPT

These attributes - if set - are processed with the help of the *Statement Manager* that will be invoked by a *GroupTransfer* object.

There are methods to manipulate, i.e. set and reset attributes in addition to services to serialise and deserialise a *Security* data structure's contents (see Chapter 12 for more information on serialising and deserialising).

9.2.6 TransferData

This class defines the actual data stored within the nodes of a container that is transmitted from client to server or vice versa. Besides a few methods to manipulate/serialise its contents the main method to instantiate a data structure is: *TransferData()*, the Class constructor which creates the data structure to be stored within a node.

10 Service Blocks on the Transfer Layer

The transfer layer provides services for payment, securing information (statements) and handling of certificates. In the following the three service blocks providing these services are described.

10.1 The Payment Block

The payment block is part of the transfer layer of *SEMPER*. The transfer manager uses the payment block to carry out transfers of money.

Like in any generic service block in *SEMPER*, the payment manager in the payment service block will utilise "modules" (external payment systems) to implement the payment services. A user may have several different modules installed on his computer - possibly implementing different types of payment systems (see [JW96, PAW96] for more information on this). A specific instance of a means of payment available to the user is abstracted by a *purse* object in the *SEMPER* payment block. A user may have several purses corresponding to a single payment module.

We classify the entire range of services provided by the payment block into five classes:

I. *Value Transfer services* constitute the core of any payment system:

- Moving electronic value from a payer's purse to a payee's purse. The payer might request a receipt from the payee or from the acquirer, depending on the selected payment model and level of security.
- Moving electronic value back from payee to payer, i.e., in reversal payments.
- Converting real money into electronic value ("loading") or vice versa ("deposit"); this is relevant only in the cash-like model.
- Moving electronic value between a user's purses, e.g., currency exchange, (maybe also converting from one payment system to another). Note that currency exchange is not a unilateral process but requires running a protocol with the bank.

II. *Purse management services* allow the user to set-up, configure, manage, and delete purses. Purse management includes access control to the purse. Some purse operations might require authorisation for each critical operation (e.g., to insert a smart card and enter a passphrase for each withdrawal) while others may not have any access control.

III. *Transaction management services* allow transactions to be queried for their status,

IV. *Information services* permit the retrieval of information on the state of the payment manager, or a specific purse: e.g., a list of previous operations, the last statement of account received, or statistics on all payments received and made in a certain period of time.

V. *Recovery and dispute management services* support the user in case something has gone wrong. This includes services to recover from a crash of the user's

system or an interrupted atomic operation, but also to retrieve or verify proofs of evidence (like signed receipts or statements of account) that must be shown to a third party.

In the following, we describe the various data structures and protocols used to provide the above services. Not all of the above services are provided by the current implementation; in particular, details of the recovery and dispute management service are yet to be worked out.

The payment service block provides, to the extent possible, a generic payment service based on a unified model that covers, hopefully, all the different payment systems available. The value transfer services are described in terms of a hierarchy of interfaces. The root interface defines common payment services expected of a purse of any payment system. Sub-interfaces define additional services provided by different *payment models*. Currently, a cash-like payment model and an "account-based" (representing credit-card and electronic cheque systems) payment model are supported. Additional models can be included later. To incorporate a specific payment into the payment block architecture, a system-specific adapter mapping the services of the payment system to the interface of one of the defined payment models needs to be implemented.

As a matter of principle, the payment manager will always try to use the services provided by the actual payment system. Only if a required service (e.g., "receipts" or "reversal") is not available will the payment manager execute a generic protocol or raise an exception.

Figure 19 illustrates an example instance of the payment block.

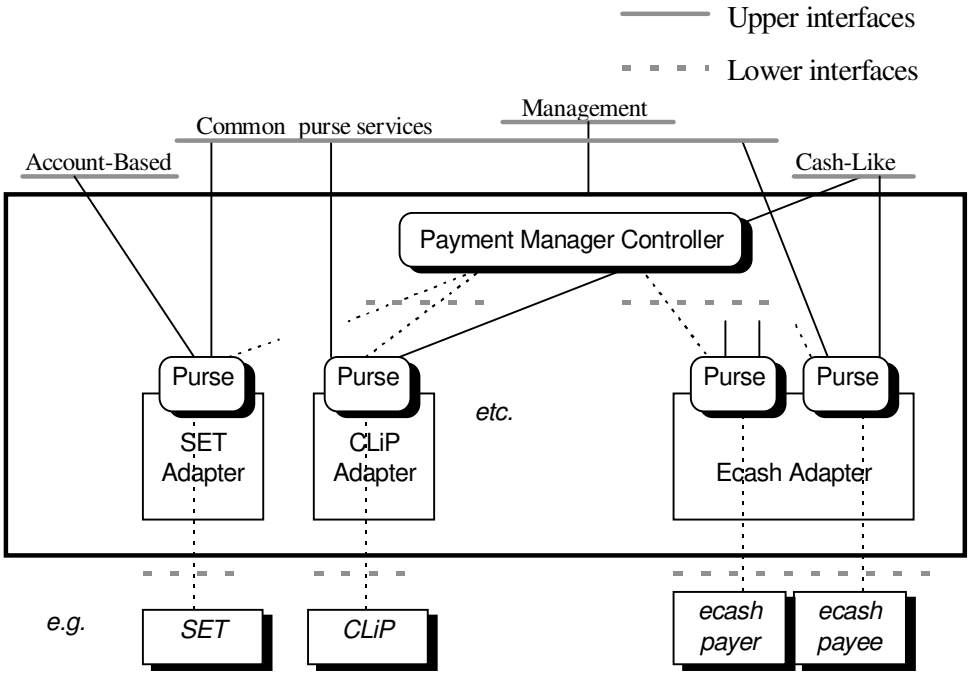


Figure 19 Payment Block Architecture

10.1.1 The Manager

The payment manager provides the following services:

- Management, control, and information

- Purse selection

Management, Control and Information

The payment manager keeps track of:

- currently active transactions,
- currently available purses,
- known payment module adapters, etc.

To maintain and manage this information, the payment manager provides various services such as creation and registration of a purse, deletion of a purse, registration of a payment module adapter, registration and de-registration of active transactions. Additional services are provided to make this information available to other objects and applications in a variety of useful ways.

The manager is also responsible for initialising the entire block on start-up. Eventually, it will provide services for shut-down and crash recovery.

Purse Selection

Selection of a purse to be used in a transaction is based on several factors:

- negotiation with the peer,
- requirements for the transaction (e.g. security requirements),
- static user preferences, and
- manual selection by user.

Apart from the first all factors are local. The payment manager provides various services to facilitate this local selection.

Negotiation with the peer for selection of the payment instrument can be done in several ways. As a beginning, we use a simple approach. All negotiation protocols consist of simple request-response exchanges. Negotiation takes two forms:

1. The initiator proposes a list of payment means and associated amounts; the responder picks a suitable subset. Figure 20 illustrates this type of negotiation. It shows both the interface events (*requestPSChoice* and *choosePS)* and the protocol flows.
2. The initiator selects a single purse and asks the responder whether the corresponding payment system is acceptable; the responder sends a confirmation. If the payment system is acceptable, the responder's payment manager selects a suitable purse for the subsequent transaction. This is illustrated in Figure 21.

The first-form is for the negotiation of payment systems whereas the second form is a hand-shake process. Both are optional -- if all the information about the payment endpoints are known a priori, then no negotiation is necessary. If the first form is used, the handshake will almost always be necessary.

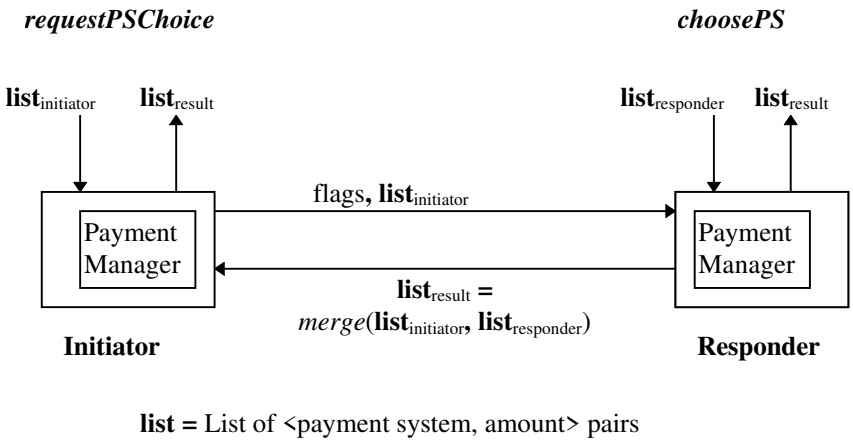


Figure 20 Negotiation protocol for selection

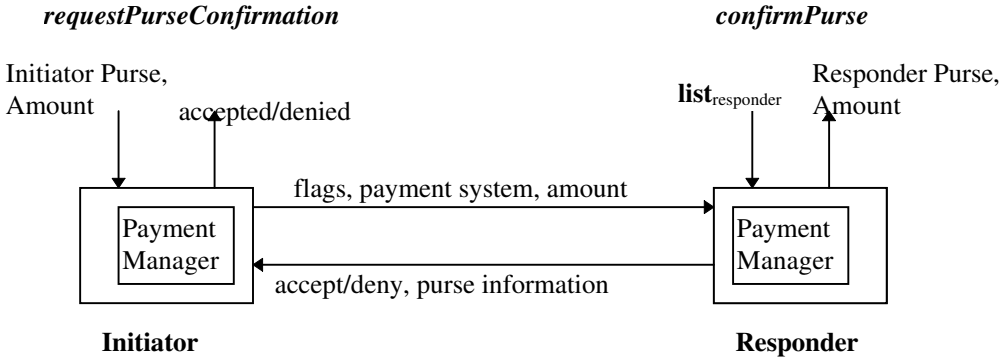


Figure 21 Negotiation protocol for confirmation

Flags in the initiator's request messages indicate the nature of the selection sought from the responder (e.g. the responder may be allowed to reset the proposed amount).

Using these services, various negotiation policies can be implemented. Methods implementing purse selection based on a default policy is provided as an example.

10.1.2 Purses

As mentioned, a purse is an abstraction of an instance of a payment system that is available to the user. A purse should provide:

- a constructor to instantiate a purse object,
- services identified in the management interface (e.g. answers to questions like "does this purse provide non-repudiable receipts for payments?")
- a set-up service which will be used by purse management applications, and
- an initialisation service using which it could be activated during start-up.

The root Purse class provides default implementations of some of these services. Adapter-writers should override/extend these implementations as necessary.

Additionally, the Purse class hierarchy also provides services for information management corresponding to these purses.

10.1.3 Transactions

The value transfer services of the payment block are described by the PurseServices interface hierarchy. As mentioned, the root interface defines services that are common to all payment models. Some example services defined in this interface are:

- *pay* makes a payment from a purse to a designated recipient,
- *receivePayment* is the counterpart of `pay`; it receives an incoming payment.

Model-specific sub-interfaces may define additional services. For example, the sub-interface for the cash-like model has a service to *withdraw* money from the bank into the purse.

Every instance of a value transfer service is abstracted by a transaction object. The transaction class implements the value transfer services described in one branch of the PurseServices interface hierarchy. A transaction has two kinds of information associated with it: transient information (such as its state) that is relevant only while the transaction is active and "permanent" information about the transaction (such as receipts or other evidence) that is relevant long after the transaction is completed. The former is kept in the Transaction object itself while the latter is kept in a related TransactionRecord object. The former will be used in crash recovery while the latter will be used in dispute management as well as for informational purposes.

The root Transaction only defines general transaction services like trying to abort an on-going transaction or retrieving its current status.

10.1.4 Modules and Adapters

To adapt a payment system to *SEMPER*, the minimum required is as follows:

- a Purse subclass, implementing the internal management interface corresponding to a specific payment model
- a Transaction subclass, implementing the purse services interface corresponding to the same payment model

In addition, a subclass of the TransactionRecord class may be implemented, if necessary.

10.2 The Statement Block

The Statement Block provides all cryptographic services needed for secure transfer of messages between the participants of an open marketplace. The manager provides services for setting up a session with its peer, and for managing the Statement Module. Setting up a session requires negotiation of parameters which will be fixed during the session and if needed the exchange of a session key. The latter is taken care of by the Statement Module.

The two essential datastructures are Statement and StatementTransaction. Basically the Statement Block offers services on instances of Statement. A Statement initially contains a message. Any statement can be encrypted (symmetric or asymmetric encryption), signed or protected by a message authentication code (MAC). A StatementTransaction object offers the same services, but the interface is much simpler as the keys and other parameters needed as input to the corresponding Statement operation are now determined by a negotiation taking place when a

StatementTransaction object is instantiated. The actual cryptographic operations on statements are performed by the Crypto Block (see Section 11.2).

In the following the manager, module and two main datastructures Statement and StatementTransaction are described in more detail.

10.2.1 The Manager

The manager is implemented as the class StatementMan. It basically provides three services:

init()
Loads the StatementModule (identified by input parameter) and builds the preference groups needed by the Statement Manager if these could not be retrieved from the archive. Initialisation of the Statement Manager requires that Archive and Preference managers have been initialised.

openInitiatorStatementSession()
Opens a statement on the client side. In this process the parameters needed for the session are negotiated. Currently the following parameters are negotiated:
- Choice of conventional crypto system
- Choice of MAC algorithm
- Method for making signatures
- Method for verifying signatures
- Method for public key encryption
- Method for public key decryption
- Hash functions
- Method for key exchange

The client proposes a number of candidates for each parameter by considering the supported algorithms, the preferences set by the client and the available keys and certificates. The server makes a similar proposal and during negotiation the first candidate from the client which is on the server's list is accepted. Other ways of negotiation are conceivable, but not implemented yet.
As a parameter to openInitiatorStatementSession the caller can indicate which parameters are needed in the session. Only these will be negotiated. After negotiation the keys corresponding to the chosen algorithms are recovered (using services from Archive and Certificate Manager)
Next, key exchange is performed (if requested by both parties), by calling a service of StatementModule as described below.
Finally, a new StatementTransaction object is instantiated using the recovered keys and negotiated parameters.

openResponderStatementSession()
This is the server side of opening a session. It works according to the description of the client side.

10.2.2 The Module

The Statement Module provides services for exchanging a key between client and server. Currently two methods are implemented, a simple key exchange in which the client chooses a key and sends it to the server, and a method in which the two parties select a key mutually at random. The two methods are depicted in Figure 22 and Figure 23.

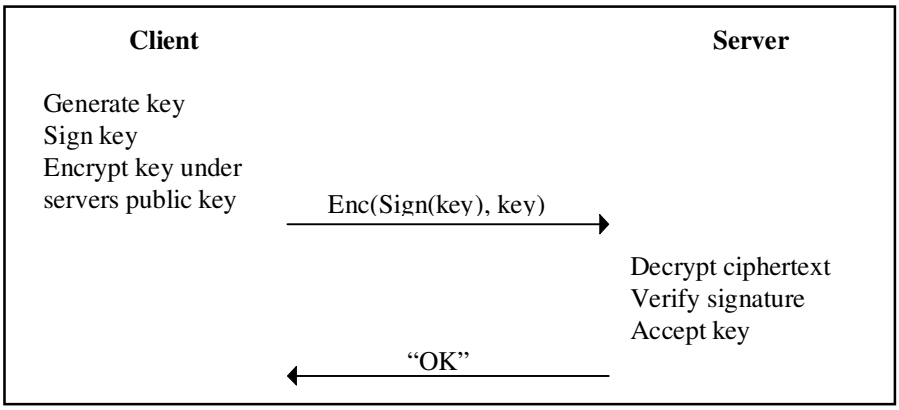


Figure 22: Simple key exchange used in Statement Module

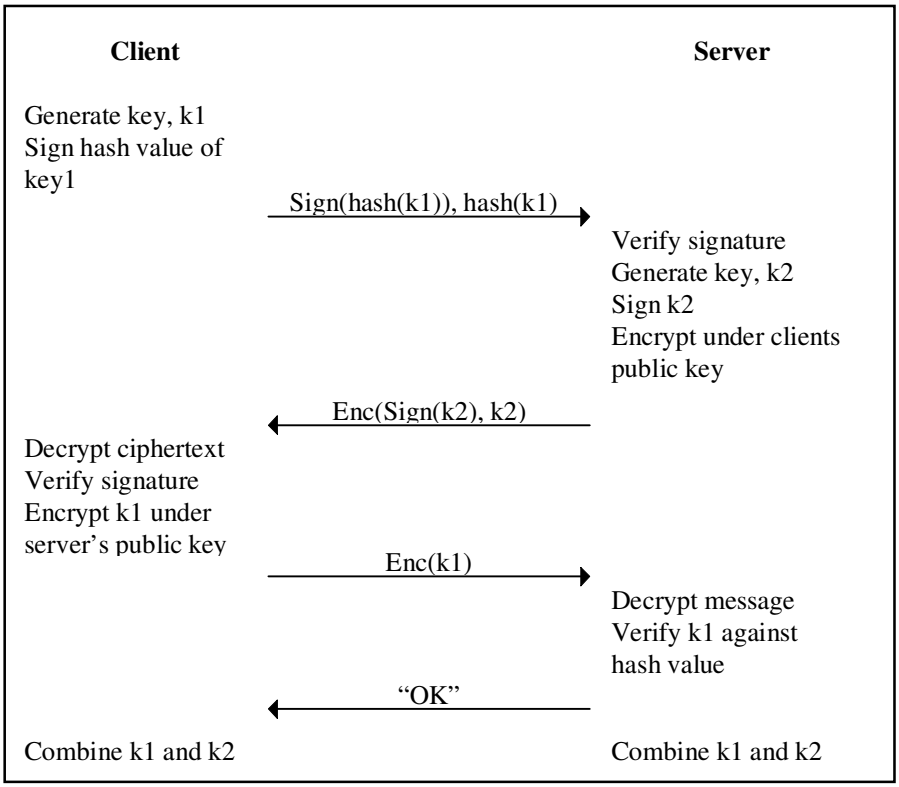


Figure 23: Selecting a key mutually at random.

10.2.3 Informal Description of Datastructures

As mentioned above the two essential datastructures in the Statement Block are Statement and StatementTransaction.

Statement

A statement contains the following attributes

- message_type;
  Indicates the form of the statement. The statement can either contain a message, a cipher text, a signed message or a MAC protected message.
- message;
  Depending on message type this field may contain the message of the statement or an externalised version of the Statement on which an operation has carried out.
- date
  The date of the operation (not always needed). Currently the date is read from the system. Later it should be possible to add time stamping services.
- senderId
- recipientId
- keyInfo
  Contains information on the key used (e.g. identifier of key used for the cryptographic operation).
- certificate
  In case of signatures this field may contain a certificate of the public key corresponding to the secret key used to generate the signature.
- cryptoInf
  Additional information about the cryptographic functions. E.g., designates hash function, padding of messages before encryption, MAC algorithms, padding of keys when encrypted under public key.
- securityVal
  Contains the cryptographic security attribute, such as signature, MAC value, cipher text, etc.

The following operations are available on a Statement.

- Generate/verify signature
- Encrypt and decrypt using a symmetric crypto system
- Encrypt and decrypt using an asymmetric crypto system (this means that a symmetric key is generated, the Statement is encrypted under this key and finally the symmetric key is public key encrypted).
- Generate/verify MAC.

All operations follow the same principle: The attributes of the Statement is encoded in a byte array (simply by linearising the object) and the operation is performed on the resulting array. This allows several operations on the same object.

All methods above are prepared to work in two modes: either with the message in RAM or with the field message indicating that it points out a file. Currently, however, only the operations with the message in RAM are fully implemented.

StatementTransaction

All the above operations on a Statement can also be performed by calling the corresponding operation on a StatementTransaction object giving only the Statement as parameter. The selection of the necessary keys is done by using the context of the StatementTransaction object (recall that these parameters were negotiated and selected when the session was initiated).

10.3 The Certificate Block

The certificate block provides services needed for registering users, issuing certificates and managing these. The block depends on a RCmodule, which provides the actual implementation of certificates and methods for registration and maintenance of certificates.

As mentioned in Chapter 6, several roles can be identified in this context:

Registration Authority (RA)
Is responsible for checking the identity of users that want to register. The importance of a RA varies with the level of trust needed. In the most trusted case the role of the RA will be filled in by a notary. Within *SEMPER*, we will at least for the first field trial assume that the RA is the same entity as the CA.

Certification Authority (CA)
Is responsible for the issuing and maintenance of certificates belonging to users and service providers. In the certification procedure a user will at some point receive a valid certificate. The CA needs a RA to check the identity before a certificate can be issued.

Directory Authority (DA)
Is responsible for the availability of certificates. In the first field trial this function will be integrated with the role of CA. The certificates will be stored in the local archive after they have been received from the CA. In later field trials a X.500 directory will be introduced and the role of DA may be separated from the role of CA.

Service Provider
The entity that provides the service for which some kind of registration is necessary. The Service Provider is responsible to choose which RA and which CA it wants to use. It is also possible that the Service Provider itself is a RA and/or CA for its services. It is up to the user if she wants to trust this.

User or Customer
The entity that wants to use services of the Service Provider. Before using the services, the user needs to register.

In the first implementation of the certificate block RA, CA and DA are unified. Furthermore, we only consider the case where a user wants to get a certificate on a key (thus attribute and hybrid certificates are not supported in the first version).

The two essential data structures are CertificateMan and Certificate. CertificateMan is the manager providing all services needed for handling certificates. Certificate is a class

defining the interface, which actual implementations of a certificate must satisfy. In the following we first describe Certificate, and then show how a Certificate can be used through the manager. Finally, the module is described.

10.3.1 Certificate

It is up to the provider or the registration module to define the attributes of the certificates. It should however, be possible to ask for the following values related to a given certificate (see the Appendix B for more details):

1. Distinguished name of CA
2. Distinguished name of subject (user)
3. Serial number for certificate (e.g. using a local numbering within each CA)
4. The public key
5. Scope (or application) of the public key
6. Type of Certificate
7. Time stamps (start and end of validity period, time of revocation if applicable)
8. Status of the certificate.
9. Reason for revocation (if applicable).
10. Signature of the CA
11. Signature and hash algorithm used algorithm

Scope

The scope is used to describe, which applications the key is good for. This could be
1. For signatures
2. For identification
3. For key exchange
4. For encryption

Types

The three possible types of certificates are:
1. Key certificates (containing only key(s))
2. Attribute certificates (without keys)
3. Hybrid certificates (with both a key and an attribute)

Here a key certificate is a usual certificate binding a key to a user. An attribute certificate is a certificate binding a particular scope to the user. Such certificates can for example be used to provide electronic authority or proxy to a user (the actual description of the proxy would then be available as part of the scope). In a hybrid certificate the two certificates are combined thereby linking a partiular public key to a particular application.

Status

The status describes the level of trust that one should have in the certificate. This depends on the registration and the history of the certificate. The status can be one of the following:

1. Notary registration: The user has visited a notary. The notary checks the identity of the user. The users signs a contract in which the public key is also mentioned. With the contract, the CA can finish the registration.
2. Registration by Mutual Contract. The user and service provider identify each other, and sign a written agreement. The CA receives a copy of the contract.
3. Letter Registration. The RA sends a letter to the customer which contains a passphrase. The customer needs to type in the passphrase during registration. Only the customer that received the letter can register.
4. Phone-Call Registration. During registration, a random passphrase is generated by the client software. This is sent encrypted to the RA. The RA checks the passphrase by calling the user.
5. The Point-and-Click Registration. This a fully electronic version. No checks can be done except verifying the e-mail address.
6. Not finalised
7. On alert
8. Revoked

All certificates with status 1-5 are in principle valid, but the correctness of the registration form has been verified with varying thoroughness.

Reason for Revocation

As mentioned in the FAST report (see [fast]) the reason for revocation can be
1. The user's secret key compromised
2. Request for cancellation
3. Change of registered credentials of user
4. The user has been swindling (proved)
5. False verification of the user at registration
6. Compromise of secret key of CA
7. Termination of CA

10.3.2 The Certificate Manager

After the initial registration handling of certificates comprises methods for
- Issuing certificates
- Verification of certificates with respect to a so called trusted public key from a CA (depending on the already known certificates and the user's preferences, this may involve on-line verification at the DA);
- Getting certificates from the DA
- Requesting and making changes such as extension, renewal and revocation

All processes between user and CA are client-server processes, in which CA simply handles any request that a user may have. In the following the information flow needed for the above process is discussed.

installTrustedPK()
This is a function by which the user can install the key of the CA. Initially, this key could come on a disk or be downloaded from a WWW page of the CA. To provide trust in this key it should be possible to show (a digest of) the installed trusted public

key on the TINGUIN. The user can then compare his digest with other publications of it.

sendTemplateRequest()
This function allow the user to send a request for a template (registration form). The server will return such a template. As part of this template the user gets a serial number, which follows the rest of the registration and ends up as the serial number in the certificate.

sendFilledTemplate()
After filling out the template the user returns it to the server, and gets an acknowledgement back. As part of sending the template the user sends a signature with respect to the public key, thereby showing the ability to use the corresponding secret key.

sendPublicKey()
This provides an alternative way of sending the public key. During registration the RA may need the key from a source other than the registration form. This need not, however, be by using this function (e.g., a hash value on a paper form could also do).

checkRegistrationStatus()
This allows the user to check how the registration is processed. The status is returned. If the registration is finalised, the user can call

getRegistration()
This simply returns the information, which the CA has registered about the user. The user can store this in archive by calling *installRegistration()*.

issueCertificateReq()
Here the user asks the CA for a certificate based on a registration done earlier. A certificate of some status code is returned.

installCertificate()
This functions store the certificate in the archive.

getCertificate()
Using this function the requester can get a (Vector of) certificate(s) given partial information on a certificate.

verifyCertificate()
This function verifies the certificate against the trusted public key. If necessary this method calls the CA for verification.

sendChangeCertificateReq()
This function allows the user to request the CA to change the certificate in a particular way. Among the possible changes are:
- Extension of validity period
- Revocation of certificate

10.3.3 The module

The RCmodule must provide an implementation of Certificate and methods for

- filling out templates (possibly using the TINGUIN),
- make registrations,
- store/retrieve registrations
- issue certificates
- verify certificates
- change certificates

The required interface is described in RCModuleInterface (see the appendix).

10.3.4 Functionality to be Added

This version only supports key certificates. Future versions will provide attribute and hybrid certificates as well. Presently there is no method for verifying the trusted public key.

11 Supporting Services

This chapter describes the services supporting the block on the upper layer. This comprises: communication, crypto, archive, preference handling, access control and the trusted user interface.

11.1 The Communication Block

The Communication Block provides connectivity between modules in separate entities. The basic communication service is provided by ComPoints. ComPoint is an object which allows other entities to access the underlying communication protocol, currently, tcp, http or mail. This service is refined by Channel, which allow the receiving module to specify the correlator of the message that it is expecting. ComPoint and Channel APIs are based on the synchronous, i.e. blocking, model.

11.1.1 ComPoints

ComPoint, like socket, allows an entity to access the underlying communication protocol. For each supported protocol, there is a class that implements ComPoint. These implementatio classes are not known outside the communication manager. Instead a ComPoint object is obtained through the communication manager, which given an address examines the protocol field in the desired address and constructs a ComPoint object of the corresponding implementation class. The addresses themselves are defined by ComPointAddress.

Two modules or managers can exchange messages through ComPoints as follows. Let the initiator be the module sending the first message and the responder the other module. The sequence of events is:

1. The responder opens a server ComPoint
2. The initiator opens an initiator ComPoint.
3. The initiator sends a message.
4. The responder, invoking accept on the server ComPoint, obtains a responder ComPoint for sending and receiving messages.
5. The responder receives the message and sends a response.
6. The initiator and responder may exchange more messages and responses.

Note that the server ComPoint must be opened before the initiator one. Furthermore, it is not allowed to open more that one server ComPoint on the same host and port.

Addressing

The ComPointAddress class provides an address object that unifies addressing across tcp, http and mail, the communication schemes which are currently supported. ComPointAddress contains the following attributes:

protocol - indicates the communication protocol: tcp, http, or mail.

host name or *mail address* - in the case of tcp and http, the name of the remote host and, in the case of mail, the mail address of the remote peer.

port - the server ComPoint port number.

path - the path on the remote host.

return address - the return address, used in the case of mail.

The ComPointAddress class provides various methods for creating instances and getting and modifying attributes.

Common Interface to Communication Protocols

The ComPoint class provides a unified interface for accessing the underlying communication protocols. The following instance methods are defined in this class:

write message - sends the message through the ComPoint.

read message - waits for and returns a message received on the ComPoint.

accept - waits for and returns a responder ComPoint when a message arrives.

get - gets the value of certain attributes, defined for various attributes.

Communication Manager

The ComManager class provides the static methods for creating ComPoints, based on the protocol field of the input address, so that the classes which implement the ComPoint interface for specific protocols need not be accessed directly. The ComManager class provides two static methods:

init - registers the protocol-specific classes and initialised them.

create ComPoint - creates an initiator or server ComPoint based on the protocol of the address.

Mail Receiver

The ComPointMailReceiver class provides a means for forwarding a mail that has arrived at a host to the communication block. It consists of a main method with a single argument, the name of the file containing the mime-decoded mail.

11.1.2 Channels

The Channel class refines the communication services of ComPoint. It allows more than one pair of modules to use a specific port number between two hosts. This is achieved through establishing channels between initiator and responder modules. A channel between two hosts is identified by a port number together with a correlator string. The sequence of events is as follows:

1. The responder opens a service-point thread at a specific port.
2. The initiator module requests a channel for a specific host, port and correlator. A channel-setup message is sent to that host and an initiator ComPoint is returned without blocking.
3. If the responder module has already requested such a channel, its request is blocked until the channel-setup message arrives. Only then, the request unblocks and a responder ComPoint is returned.

4. If on the other hand, when the channel-setup message arrives, no such channel is requested, then the setup message waits for a request. When a responder module requests the channel, a responder ComPoint is returned immediately and the channel is set up.

The only sequence constraint between the initiator and responder code executions is that the service-point thread must have been started before the initiator opens the channel. Also, it is not allowed to open more than one service-point thread on the same host and port.

Note that the channel class not only provides multiplexing over individual port numbers, it also allows the responder module to act after the initiator module.

Channel Handler

The Channel class provides static methods for starting service-point threads and opening initiator channels:

start service-point thread - starts a service-point thread for a specific protocol and port.

initiator open-channel request - sends a channel-setup message for a host, port and correlator and, without blocking, returns an initiator ComPoint.

Channel Service Point

The ChannelServicePointThread class provides two functions. It defines a method for responders to indicate that they are waiting for specific channels. It also processes each incoming channel-setup messages, matching it to a waiting responder, if any, otherwise, buffering it. The corresponding methods are:

responder open-channel request - waits for a channel with a specific correlator.

main body of the thread - matches incoming channel-setup messages and responder open-channel requests.

11.2 The Crypto Block

The Crypto Block provides all cryptographic services, which are needed for secure services in *SEMPER* (note, that a module may use its own cryptographic tool). The crypto block is entirely used as a supporting service and does not provide a transaction class allowing the creation of crypto sessions.

The two essential data structures are CryptoMan and CryptoKeyInfo. CryptoMan is the manager, providing the cryptographic services, while CryptoKeyInfo is used to contain key material. In the following the manager, module and CryptoKeyInfo are described in more detail.

11.2.1 The Manager

The crypto manager has a state consisting of the choice of the crypto module, the state of the module and the user specific master key. The state of the module is mainly the state of its pseudo random generator. The master key is used for protecting other keys (see Section 11.2.3). In the current version the master key is a DES key.

init
Loads the CryptoModule (identified by input parameter) and initialises the (pseudo random generator of the) module. The very first time *init* is called a random string (seed) of at least 16 bytes/characters must be provided. This seed is stored in archive and will be updated and used in all subsequent initialisations. This updating can be done in two ways. If a seed is supplied as a parameter, the old seed is combined with the new one (each byte of the old and new seed are simply added). If a seed is not supplied, the old seed is updated based on a reading of the time. This ensures that the same seed is never used twice for initialisation of the module.
Another part of *init* is to load the master key from archive and decrypt it (it is encrypted under a password) so that it can be used in subsequent calls of the services of the manager. The very first time *init* is called this is not possible. In this case a new master key is generated and installed.

masterKeyEncrypt and *masterKeyDecrypt*
These services provide encryption and decryption of other keys, respectively, under the master key. They will normally only be called in order to protect keys and to recover keys before usage. As mentioned above encryption and decryption under the master key presently use DES.

Encrypt and *decrypt*
Provides conventional encryption and decryption. The key supplied as input designates the crypto function to be used.

generateHash
Returns the hash value of the input string using a hash algorithm given as input

generateMAC and *verifyMAC*
Generates a MAC (message authentication code) of the input. The choice of method is partly described by the key, partly by an additional parameter. For example, there are standard MAC methods based on block ciphers. In this case the key will designate the block cipher, while the additional parameter will designate the particular method.

generateSignature and *verifySignature*
Generates and verifies a digital signature. The signature algorithm is described by the key while the hash function to be used is given as parameter. Note, however, that if DSS [DSS] is used, SHA-1 (see [SHA1]) will automatically be used as hash function.

For all the above algorithms it is possible to set a flag in the input, which indicates that the input text is not in RAM but on a file. In this case the input message is the name of the file.

pkEncipher and *pkDecipher*
pkEncipher encrypts an input message under a public key. Normally this message is a key, and therefore this function only works for messages which are not longer than the allowed block size for the key (for RSA this is the modulus length). If no input message is supplied, this message will generate a conventional key (for a given system) and encrypt this key. Both the key and the encrypted key will be returned. This particular functionality supports the very useful combination of symmetric and

asymmetric encryption. pkDecipher simply decrypts the input under the given secret key.

generateSymKey
This method generates a CryptoKeyInfo object containing a new key for the symmetric crypto-system given as input.

generatePubKey
This method generates CryptoKeyInfo objects of corresponding public and secret keys. See Section 11.2.3 for a description of the structure of such pairs.

randomBytes
This function gives direct access to the pseudo random generator of the module as it simply returns a requested number of pseudo-randomly generated bytes.

11.2.2 The Module

The cryptographic module provides an API supporting the functionality presented above. The only change in the use of CryptoMan and the module is that the manager must unpack (decrypt) all keys from their CryptoKeyInfo object and supply the key as an array of bytes to the module. The encoding of keys into arrays depends on the type of the key and is not described in this report.
In practice this means that a module to be used in *SEMPER* must supply an adapter converting the particular key encoding used by CryptoMan to the particular internal key encoding of the module. Furthermore, the adapter must be able to understand the encoding of algorithms as integers. This encoding is defined in CryptoConstants.

11.2.3 CryptoKeyInfo

CryptoKeyInfo is the central class for handling keys and it has the following variables:
- Key identifier
- Actual key value (this may be the encrypted key or it may be empty - e.g., after storing the key in archive)
- A flag indicating if the key value above is encrypted or in clear
- An algorithm description indicating the algorithm for which the key can be used.
- A capability used to control access to the key
- A pointer to another CryptoKeyInfo object.

The purpose of the first four variables should be clear. In the following the other two variables are discussed and the methods for protecting and storing/retrieving keys are described.

At any point in time it is possible to define a capability controlling the access to the key (the service *setCap*). This means that in order to use this key (in order to get the actual key value) this capability must belong to a role which has been opened (see Section 11.4). Note, that the CryptoManager is not responsible for associating capabilities to keys. If necessary, this must be done by the entity generating and using the key.

The pointer to another CryptoKeyInfo object is a storage reducing facility. In some crypto systems (e.g., DSS) a key can be split in three parts:

- A public key common to many users (in DSS this is the description of the groups and a generator)
- A user specific public key
- A (user specific) secret key.

For such systems, the CryptoKeyInfo object corresponding to the public and secret key will contain a pointer to the common key.

The two most useful functions on CryptoKeyInfo are getKeyVal and storeKeyVal.

storeKeyVal
This function actually provides two functionalities:
- Encryption of the key
- Storage of the key

If encryption is requested the key will be encrypted under the master key (this is default, see Section 11.2.1) or under a key supplied as parameter.
If an expiration date is given as parameter the key is stored in the archive. After being stored, the key value may be deleted from the object using *setKeyVal(*null*)*.

getKeyVal
The actual key value is returned according to the following rules:
- A key value can only be returned if the capability controlling the key is unlocked or can be unlocked.
- If the key is in RAM it is decrypted (if necessary and requested)
- If the key is not in RAM a CryptoKeyInfo object with the same key identifier is searched for in the Archive. If such an object is found, the key value from this is returned (according to the above rules).

Retrieval of a key requires the key identifier. This identifier is assigned, when the key is generated. The crypto manager does not maintain a list of such identifiers. These must be stored by the entity using the key (the key identifier can be read using *getKeyId*).

11.2.4 Functionality to be Added

Future versions of the crypto manager will provided methods for changing the master key.

11.3 The Preference Block

The Preferences Block provides a unified interface to the management of *SEMPER* preferences and configuration information. Inside the Preferences Block, configuration information represents an almost static datastructure *SEMPER* needs at start-up. This includes all information specific to a particular installation like the target system, the memory size, the screen type or any other hardware devices currently supported. Usually the configuration information is changed at *SEMPER* installation or when a new module is added to the system. Although configuration data are quite static, the Preferences Block provides services to change them. Preferences represent more dynamic data which can be observed and changed by the user at any time. Configurations are usually retrieved from a standard file which can be found on the user storage device, whereas preferences are stored encrypted in the Archive Manager.

Each module/manager defines it's own set of preferences by grouping together several fields. The datastructure allowing groups of fields manipulation and definition is called PrefGroup. Fields are defined by the PrefFields datastructure. In addition to its actual value and label, each field also defines its graphical appearance which is used for interactive editing.

11.3.1 User Configuration

Configuration data have a very simple structure defined in the Configuration class. This is a collection of configuration fields each having a name described by a string. Associated to each field, its actual value is represented by another string. Methods *put* and *get* are used to put a new field and get the value of a field in the configuration database. Some input/output methods are also available. All of them use the local file system because configuration data have to be available before the Archive manager is initialised. We next describe the main functions provide by the Configuration class:

init
Before using any of the Configuration services, initialisation must done. A call to *init* loads into internal representation the default configuration file "_sempconf". It is also possible to ask the *init* method to load user configuration from another file.

load
This can be used to load the default configuration file from the local file system. It is also possible to load a configuration file saved on a different name.

save
This is used to save in the default configuration file the current user configuration setup. It is also possible to save in a different file than the default one. This function is also used to build a new configuration file from which the Configuration can be initialised.

11.3.2 The Manager

The Preferences Manager is implemented as the class Preferences. It provides services for initialisation, input-output and groups manipulation as described in the following.

init
Configuration must be initialised before Preferences, otherwise the Preferences Mananager refuses to be initialised. Preferences initialisation loads the preferences environment from the default file stored in the Archive. If no user preferences are found, the Preferences Manager is initialised with an empty set of preferences. This initialisation process only requires the availability of the Access Control in order to be completed. Additionally, the Archive Manager is needed if a non-empty preferences environment is expected.

loadPreferences / savePreferences
Once initialisation is done, the Preferences Manager can load a preferences environment previously saved in the Archive. The *loadPreferences* method can be used when initialisation has not succeed to retrieve a preferences environment.

editGroup
This allows interactive editing of a preferences group. This service is closely related to TINGUIN. Calls to *editGroup* are usually made by TINGUIN which gives access to the editing of each group in the current environment. The Preferences Manager allows simultaneous editing of different groups. This allows the user to look at different groups of preferences simultaneously.

addGroup / delGroup / getGroup
These services give the basic tools to manipulate groups of preferences. The first two services are also used when a module/manager needs to build its group. When the business application does not succeed to retrieve an environment, each module/manager should be able to construct its group with default values by adding it to an initially empty environment. To make preferences available, the new environment must be saved.
The service *getGroup* is used to retrieve all data contained in a group. This is used in particular when managers are initialised.

11.3.3 Preferences Groups

The Preferences Manager processes groups of preferences fields. Each group is implemented as PrefGroup. A PrefGroup processes preferences fields in the same way as the Preferences Manager does. The following methods are used:

load / save
This allows to load and save individual groups in the Archive. This can be used to build a preferences environment. This is not fully implemented yet.

get / put
These are used to get or put a particular field from a group. Fields are indexed by their name. The order in which fields are put is important because of the way formatting is processed (see below).

11.3.4 Preferences Fields

A preferences field is the atomic data structure for preferences. Each field is implemented by a PrefField object. These objects have a current value and an appearance. The following field appearances are currently supported:

- *Label*: Is a string displayed in the preferences group.
- *String*: Is an input string with a current value.
- *Checkbox*: Is a check box with a description of its purpose.
- *List*: Is a list of items with one of them selected.
- *NegotiationList*: Is a set of items from which some are unused and some are part of a priority list.

We describe below some services provided by every field

getValue / setValue
Allow to get and set the value of the field. The type of value depends on the field type.

showme / cancel / applyModifications / isOK

These functions provide the management tools for interactive editing. The first method shows the field in a frame which can be displayed by the TINGUIN. The user may then decide to apply the modification or to cancel changes. Changes are applied only if the new value set by the user is allowed. When applied, changes are saved in the archive as the new default preferences file.

Formatting Fields within a Group

Each field contains formatting information in order to create a user-friendly display of each group. When a group is displayed, fields are put one after the other according to formatting constraints. These contraints are relative to each other and are applied in the same order fields were put in the group. When the Preferences Manager puts a new field for display, the constraint for that field is added to the previous one. Since a constraint A followed by a constraint B is not the same as the contraint B followed by the constraint A, the order in which they are applied is important. The Preferences Manager applies constraints from the first field of a group to the last one in the same order these fields were added when the current group was defined. These formatting constraints allow flexible interaction between the user and the group appearance by supporting automatic frame resizing. Each field type comes with default formatting constraints which usually give good results. For a more complex group, fields constraints can be redefined allowing to design the desired group appearance. For more information about constraints see the *GridBagConstraints* class in the standard Java JDK.

11.3.5 Preferences and TINGUIN

The default TINGUIN window contains a preferences menu by which groups can be selected by the user. When a group is selected, the Preferences Manager starts to run interactive edition of that group. Each field appears in the window giving the user the ability to select new values. Two options are then possible:

- *Cancel* of every modification performed so far in that group editing and
- *Use* new values save them before closing the current group editing. If some values have not been set correctly, the user is prompted to correct the setting.

In addition, the preferences menu may provide choices for loading and saving a preferences environment. This integration has not been done completely yet.

11.3.6 Advanced Services

The Preferences Manager can easily be extended in order to manage more dynamic data. For instance, in addition to the set of available values for a parameter the Preferences Manager may also store the actual negotiated value. This can be useful in case the user performs the same business application frequently, as it is natural to re-use the result of previous negotiations. The management of this kind of information is not currently supported by the Preferences Manager but this can be done easily by upgrading the *PrefFieldNegotiable* field type. For instance, suppose the field, F, is a negotiable field with a set of options sorted in a priority list (this is already supported). This represents initial choices for a negotiation parameter which belongs to a given preferences group. When an application requires negotiation of a paramater corresponding to F, the first thing to do is to check if that negotiation has been done

previously. By adding some information to F, the Preferences Manager may give back the result of that previous negotiation without asking the user. If no such prior negotiation is found, the Preferences Manager could ask the user to accept or to refuse a choice. If the user accepts, the Preferences Manager records the negotiated parameter with the description of the situation in which this result happened. Next time the same parameter in the same situation has to be negotiated, the result of that previous negotiation can be used again.
In addition to the description of the situation and the resulting parameter, F must also indicate the time frame in which the parameter value can be used. After the expiry date user interaction is again necessary to conclude negotiation. This extension will be addressed in the next release of the Preferences Manager.

11.4 The Access Control Block

The Access Control Block provides all services needed to implement access control on the data and methods of other blocks. Access control ensures that potentially dangerous, compromising or undesirable operations cannot be performed by unauthorised users or without the user's explicit consent. Access control is *capability* based. Each *SEMPER* block must declare a capability for each action for which unconditional access is undesirable. Such a capability must be verified each time the corresponding action is performed. The Access Control Block provides the necessary services to manage such capabilities.
In its simplest form, verifying a capability involves asking the user whether the capability is (still) valid. To avoid an avalanche of questions when a high level operation triggers many low level ones, managers can declare *roles* that can contain many capabilities. Roles are either locked or unlocked; depending on their type they may require passwords to be unlocked. If a simple capability belongs to an unlocked role, it is considered valid. A capability may belong to several roles. If only one of these roles is unlocked, the capability is (potentially) valid. If not, it is invalid.
Just as there are different types of roles (differing in they way they are unlocked and, hence, level of security), there are different types of capabilities as well. Simple capabilities are always valid (as long as they belong to an unlocked role); other capabilities may be valid only if a pass phrase was entered that has not expired yet.
The Access Control Manager does not restrict the types of roles and/or capabilities. Instead it declares two abstract classes Role and Capability that provide the core interfaces and implement the basic functionality needed to perform access control. Some fully functional (i.e. non-abstract) default roles and capabilities are declared as subclasses. The advantages of this method is that other managers can declare other types of roles or capabilities specific to their needs.
In the following the manager AccessMan, and the two main classes Role and Capability are described in more detail.

11.4.1 Roles

A role is either locked or unlocked. When a role is locked, the user may be asked to unlock it. The method *islocked* checks whether the role is locked. The method *lock* locks the role. The method *unlock* asks the user to unlock the role (possibly asking for some form of authentication). Roles differ in the way they are unlocked. Therefore *unlock* is an abstract method making *Role* an abstract class.

All roles store a string description describing the role. When the user is queried to lock or unlock a role, this description should give the user a good understanding of the sort of capabilities likely to be stored in this role, and therefore give an indication of which actions she allows if she unlocks the role. The description is specified when creating the role.
The Access Control Manager keeps track of all roles that currently exist. These can be queried through the class method *Role.getAllRoles* (which, for example, is necessary if the user wishes to lock all existing roles). When a new role is created, a reference to it is automatically added to this database of current roles. Therefore, before destroying the last external reference to a role, it must be removed from the database using the method, *finish*. Otherwise the reference in the database would prevent the role from being garbage collected (and it would erroneously show up in the list of all existing roles).
The access control block currently provides three types of roles:

- UnlockedRole: Roles in this class are always unlocked.
- OrdinaryRole: Roles in this class can be unlocked without some form of user authentication.
- SecureRole: Roles in this class can only be unlocked when a valid password is entered.

Other types of roles are conceivable.

11.4.2 Capabilities

For each action it performs, and for which controlled access is desirable, a *SEMPER* block must declare a capability. A valid capability allows execution of the associated action, provided it also belongs to an unlocked role.
All capabilities store a string description describing the capability, from which the user can deduce what kind of action is associated with the capability when she is asked to validate it. The description is specified when creating the capability.
Capabilities must belong to one or more (unlocked) roles to be of any use at all. The Capability class defines the following methods to manage membership of roles:

- *addToRole(Role r)*: Add the capability to a role.
- *removeFromRole(Role r)*: Remove the capability from a role.
- *findUnlockedRole*: Check whether the capability belongs to any unlocked role.

The latter function is protected and for internal use by AccessMan in the implementation of *verifyCapability*. Similarly, each capability class must specify a protected method *isValid* that can be used by AccessMan to check whether the capability is (still) valid. Depending on the type of capability that implements this method, this may always return true or involve the (re-)entering of a password (if it expired).

The access control block currently provides two types of capabilities:

- SimpleCapability: Capabilities in this class are always valid.
- PassWordCapability: Capabilities in this class are only valid if a valid password was entered, and expire after a time-out.

Other types of capabilities are easily added to the system. The current implementation makes capabilities streamable.

11.4.3 The Manager

The class AccessMan sole purpose is to provide a more or less secure implementation of the method *verifyCapability* to verify whether a capability is valid. Before performing an action, a manager must call *verifyCapability(capa)*, where capa is the associated capability. This method will check whether the capability belongs to any unlocked roles, and if so, verify whether the capability is valid. If the capability does not belong to any unlocked roles, the user is prompted to unlock one of these first. If the method is still not a member of any unlocked roles, or if the capability is not valid, *verifyCapability* returns false; otherwise it returns true.

11.4.4 Using the Access Control Block

Consider a block foobar which implements two actions: one to output `foo` and the other to output bar. In this section we will show how one can guard both actions using the access control block.

Each manager must declare a few roles that will group related capabilities and provide increasing levels of security. This is best done as part of the *init* method. In this example foobar declares just two roles: foobar_unlocked_role and foobar_secure_role.

```
Public void init
{ foobar_unlocked_role = new UnlockedRole("Foobar's unlocked role") ;
  foobar_secure_role = new SecureRole("Foobar's secure role"
                                    ,"SECRET") ;
                        // with password "SECRET".
```

Next, the manager defines two capabilities associated with the actions.

```
  foo_capa = new SimpleCapability("Allow to say FOO") ;
  bar_capa = new PassWordCapability("Allow to say BAR","123",10000) ;
          // with password "123" and a timeout of 10000 milliseconds
```

And adds them to the roles.

```
  foo_capa.addToRole(foobar_secure_role) ;
  bar_capa.addToRole(foobar_unlocked_role) ;
} // end of init
```

Now the original actions are guarded with the capabilities using the following piece of code.

```
if (AccessMan.verifyCapability(bar_capa))
  System.out.println("-> BAR! ")
if (AccessMan.verifyCapability(foo_capa))
  System.out.println("-> FOO! ")
```

By virtue of the first guard, the user is requested to enter the password for the capability (i.e. "123"). The capability bar_capa belongs to an unlocked role; therefore the user does not have to unlock any role to enable the first action. To enable the second action, the user must unlock the secure role (i.e. with password "SECRET"). This time, nothing has to be done to make the capability valid, because it is valid by default.

11.5 The Archive Block

The archive block provides supporting services for local data storage and archival. It is in charge of storage and retrieval of all persistent data of the various *SEMPER* blocks which are stored for current or later use. It includes particular features for a secure storage.

The archive manager is the provider of the archive services. An underlying database module carries out all the elementary functions for adding, finding and deleting objects in a database.

11.5.1 Access Name

Every archived object must be referenced by a meaningful name which is an access key from the point of view of the database.
AcccessName is a class that includes two attributes:

- **name**
  A string implementing the meaningful name of the object. The syntax of the name is chosen by the users of the archive services. It is recommended to insert the object class name as a prefix.
- **serial_number**
  A long integer optionally joined to the name.

The accessName objects are used as input parameters of the Archive manager services. An entity which submits an object for archiving can create an access name with a serial number (see the archive manager service getSerialNumber) or without.

The accessName class is planned to provide more complete handling of archived objects in the future.

11.5.2 Storage Attributes

For each archived object some associated storage attributes must be set at the storage time. Except the attribute time of storage, these attributes are input parameters of the *add* and *modifyAttr* Archive manager services (see below).

- **Security requirement**
  Indicates the requested level of security for the object. This information is used to perform access control using the capabilities and provide the security features.

- **Availability requirement**
  Indicates the requested level of availability for the object. This information is planned to be used to provide the availability features. Used by the back-up function.

- **Expiry Date**
  The expiry date gives the date on which the archive manager is allowed to remove the archived object by garbage collection.

- **Time of storage**

This information is associated with each stored data by the archive manager in order to allow further searches according to the date.

11.5.3 The Manager

The archive manager is implemented as the class ArchiveMan.

Objects from all classes can be archived provided they implement the Streamable interface or they are standard Java classes.

Particular Features

Security
The secure storage provided by the archive manager covers the needs of confidentiality and integrity of the sensitive data. Depending on the kind of data to be stored it may require data encryption and, optionally, storage in a tamper-resistant device.
Every entity submitting an object for archiving must indicate the security level required for storage of that object. That security requirement is then fulfilled by the use of various encryption algorithms and keys or different hash functions, using the services provided by the Crypto and Access Control blocks.
The integrity control consists in checking the content of the stored data in order to be sure that they have not been modified by an unauthorised entity. To do this, the archive uses the MAC services provided by the Crypto block.

The security levels that are implemented are: none, password protected, encrypted. Storage in a tamper-resistant device as well as integrity congtrol will be implemented later.

Back-up
In order to increase the availability, selective back-ups can be performed within the archive. These back-ups are more selective than general back-up the user could do on his own initiative using the operating system of his computer.
The archive back-ups are done on user's request via the TINGUIN. The user can ask for a selective back-up: a list of items is proposed for him to choose (e.g. logs or waste files). A back-up automatic proposal can also be done to inform (or remind) the user that some important data should be saved apart.
The peripherals on which back-ups can be done are specified within the archive user's preferences and can be selected interactively by the user when asking for the back-up.

The back-up function is to be implemented in the future.

Garbage collection
This provides a cleaning of the archive database as a function of the expiry date associated with every stored data. The expiry date of an object within the archive is set by the entity that first submit that object for archiving. Out-of-date data are not definitely removed from the database but copied in waste files.Three cases are foreseen for the garbage collection. They must be selected within the archive user's preferences:

- automatic garbage collection,
- automatic garbage collection with confirmation by the user,
- garbage collection on demand, only on user's request.

Services

The Archive block uses a capability based access control with the help of the Access Control block. The capability required for the use of the archive services is fixed according to the security requirement associated with each stored object. Possible capabilities in the frame of the archive are passphrase, encryption/ decryption key, or none.
All the services are performed according to the current role's capability, except *getSerialNumber* and *log*.

It should be noted that each object submitted for archiving is serialised into a byte stream by the archive manager before being stored in the database.

init
Initialises the manager to allow the archive services to be processed. At the very first init, it creates all the files needed by the archive.

getSerialNumber
A serial numbering service is provided for the use of the other blocks. That service gives on demand a number which is unique and can be used for purposes like naming of objects.

add
Adds the given object identified by its access name to the archive. Sets the attribute date of storage. The object is stored according to the security requirement given as input parameter.

read
Searches in the archive for an object identified by its access name and returns the handle of this object.

del
Removes from the database an object characterised by its access name.

update
Searches in the database for an object identified by its access name, removes it, and replaces it by the new object with the same access name. The attribute time of storage will be updated according the current date.

modifyAttr
Updates the values of the attributes attached to the stored object identified by its access name in accordance with the given parameters. Neither the object itself nor the attribute time of storage is changed. If the attribute security requirement is modified, the way to store the data must be updated accordingly (encrypted or not).

GetSecurityReq, GetAvailabilityReq, GetExpiryDate
Gets the value of the respective storage attributes associated with a stored object: security requirement, availability requirement, expiry date.

search
Searches for all objects identifiable by the given part of access name, and returns a corresponding set of access names.
Optionally, the search may be done according a time window, by reference to the attribute time of storage associated to the stored object.

log
Adds the given object to the log file with an attribute date of storage.
Logs are done in order to perform audits. They consist in storing data which are representative of the transactions between the host entity and other entities. They have the particularity of being "write once" and are stored in a separate database. The local time of the entity is used for time stamping.

read Log
Reads the log file. This service is not yet implemented.

11.5.4 The Database Module

The database module provides elementary functions needed by the archive services. It should be noted that the Database module must only be accessed via the Archive manager.

The interface between the archive manager and the database is provided by the interface class Database which has the following methods: *store, delete, readDate, readAttr, readData, modify, traverse* (search). The parameters of these methods are:

- key: the string shape of the access name.
- partial key: a string representing the searched part of the access names.
- date: the date of storage.
- attr: the string shape of the storage attributes.
- data: the string shape of the object to be stored, after serialisation.
- startDate, endDate: dates used for definition of the time window of the search service.

DatabaseNoIndex is the class that implements the database. This particular implementation does not provide an index to the stored database, and looks up information every time a request is executed on the database.

The storage model uses the following fields:

- Entry: (entry-size, record-size, record, gap)
- Record: (key-size, key, date, attr-size, attr, data-size, data)

11.6 TINGUIN

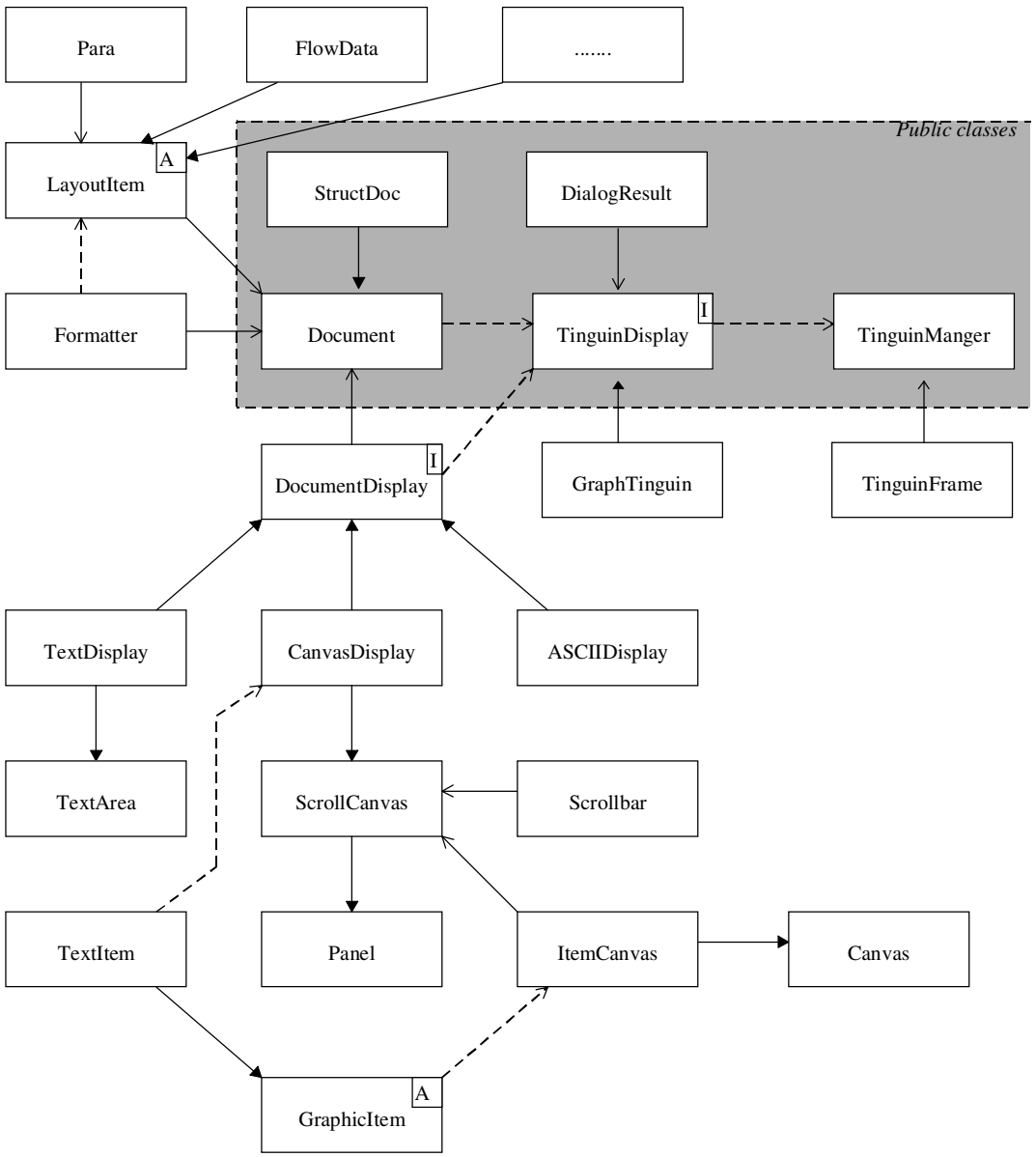


Figure 24**: Complete class diagram**

The Trusted Interactive Graphical User Interface, TINGUIN, provides a user interface to the user for all *SEMPER* managers. Each (confidential) action might need user interaction, e.g., a payment action or an order has to be confirmed. TINGUIN provides simple dialogue mechanisms to handle this interaction. The public Java interface (see Figure 24) consists of five classes:

- The TinguinManager is the entry point for the programmer. It handles the various sessions, allocates new (TINGUIN) displays or deallocates displays which are not needed.
- The TinguinDisplay is an interface class that represents a display for a (business) session. It offers mechanisms to display simple dialogues which request interaction from the user. A display is presumably shared among the different managers and therefore it has to be acquired and released after usage.

- The Document and StructDoc classes define structured text which is shown via a TinguinDisplay.
- The DialogResult class is used for returning the results of user interaction.

11.6.1 TinguinManger

The TinguinManager is a static class or, more precisely, it contains only static members. Therefore, it does not need an instantiation, although an initialisation will be necessary in order to build a menu for the preferences. The TinguinManager is used to create a new display for a session or destroy a display when the session ends. The following methods are available for this:

newSession / *closeSession*
The newSesssion method returns a reference to a TinguinDisplay (currently this is an Object of the class GraphTinguin). The session can later be closed by *closeSession*.
void TinguinManager::closeSession(TinguinDisplay displayToRemove);

buildPrefs
This method can be used to build the preferences menu.

11.6.2 TinguinDisplay

Access Control

This display is used for all outputs and inputs of the according session. Since a display can be a critical resource, access must be controlled. In order to prevent that several managers use it simultaneously, a display must be acquired. If it is acquired only one manager has access to it. After displaying something the manager should release the display as soon as possible to ensure that other managers can have access to it.

Example:

```
doc display.acquire();
display.showDocument( doc );
display.release();
```

Document Based Input/Output

Currently the TinguinDisplay provides 5 methods for displaying a document.

1. The simplest method is *showDocument(Document doc)*. This clears the current display and displays the document doc. It does not wait for user input or response.
2. *confirm(Document doc)* does the same, except that it waits for the user to press a confirm button.
3. In order to get a yes or no answer from the user the method *yesNoDialog(Document doc, String yesLabel, String noLabel)* is appropriate. If the user pressed the button with the yesLabel it returns true, otherwise it returns false.
4. If some kind of selection of an object is necessary the method *selectObject(Document doc, Vector objectList)*; may be invoked. The first argument is, as usual, the document to be presented. The second argument is a list (or a

Vector) of objects which are presented in an option list. The objects in the list will be shown using their *toString* method.
Example:

```
Vector list = new Vector();
list.add(new String("String Object"));
list.add(new Integer(5));
list.add(new Dimension(20, 20));
DialogResult r = display.selectObject( doc, list);
```

5. To request the user to fill out a simple form *evalForm(Document doc, Vector fields)* has to be invoked. Again doc is the document to be presented. The Vector field contains Strings which will appear as labels for the textfields. The method returns an instance of class DialogResult. The method result of that object returns a Hashtable containing key/value pairs. The keys are the elements from field and the values are the inputs that the user has entered in the according text field.
Example:

```
Vector fields = new Vector();
fields.add( new String( "Enter firstname"));
fields.add( new String( "Enter surname" ) );
DialogResult r = display.evalForm( doc, fields );
System.out.println("User pressed ok button:" +
r.ok().toString());
Hashtable t = (Hashtable)r.result();
for (Enumeration e = t.keys(); e.hasMoreElements() ;) {
        String key = (String)e.nextToken();
        String value = (String)t.get(key);
        System.out.println("Key was " + key + "Value was "
value);
}
```

Simple String Dialogues

In addition to the document based dialogues there are simpler message dialogues, which instead of presenting a document show a simple multi-line string message. Again these methods are elements (implementations) of interface class TinguinDisplay. To confirm a message the method

void TinguinDisplay::confirmMessage(String message, String confirmLabel);

can be used. Moreover to get a yes or no from the user the

void TinguinDisplay::yesNoMessage(Sting message, String yesLabel, String noLabel);

is appropriate. Also similar to the document based dialogues the method

DialogResult TinguinDisplay::formMessage(String message, Vector fields);

works just like the document based equivalent. The string may contain '\n"s in order to separate lines in the message.

Results from Dialogues

Dialogue results will be returned in an instance of class DialogResult. This class has two public members

ok
This method returns true if the user selected ok, yes or whatever the ok-button was labelled with, otherwise it will be return false.

result

This method returns the actual result object. If a form based dialogue such as *evalForm* or *formMessage* was used this is a reference to a Hashtable. In case the *selectObject* dialogue was used this is just the selected object.

11.6.3 Documents

Documents are reflected in the class Document and StructDoc. Presumably most programmers will use only StructDoc since the base class Document is quite low level.

In general documents contain text and layout information such as margins or fonts. The layout information is handled on a stack. So whenever a new font is pushed on the font stack it is valid since the top most font on the stack is valid. After popping a font from the stack this font is not valid anymore but, as before, the (new and old) top most font is valid.

To make the creation of documents more convenient the subclass StructDoc was introduced. This class offers methods for adding ordered/unordered list, headings with different levels, description lists and a simpler font changing mechanism. The methods are:

beginOrderedList
is used to start an ordered list.

addItem
Items can be added to the list with this method.

endOrderedList
This method closes a complete list.

beginUnorderedList / *endUnorderedList*
To handle unordered list instead of beginOrderedList / endOrderdList

addHeading
Adds a heading with a given level. The level may be in the range of 1 to 5 where 1 is the biggest size.

describeItem
To add a description of an item. This will print the item in one line. The next line contains the indented description.

beginItalic / *beginBold* / `beginFixed`
The following text will be inserted using the given font. Using *endBIF* it is possible to revert to the old font.

addFlowingData / *addLiteralData*
Adds the text given as parameter to the document.

addBoldFlowing / *addItalicFlowing* / *addFixedFlowing*
These allow adding just a single string given as parameter in the given font.

12 Implementation Architecture

This chapter deals with implementation issues such as choice of language, integration into WWW and encoding of objects into strings of bytes, which is needed for communication, storage and cryptographic services

12.1 Implementation Language

As *SEMPER* has to run on a variety of platforms it is of uttermost importance that all parts of *SEMPER* are as portable as possible. A clean abstraction mechanism to separate the different parts of the *SEMPER* architecture is a requirement for the implementation language as well as easy prototyping. The availability of (portable) standard libraries for graphics and communication is an additional criteria we measured an implementation language. Last but not least the implementation language should also offer sufficient performance.

Taking all of this into account in comparing languages such as C, C++, Python and Java we finally decided that, in order to create a uniform implementation environment, all new programs and applications for use in *SEMPER* will be written in the JAVA language.

Initially JAVA is simply used as programming language, which means that the security problems of JAVA does not affect *SEMPER* at all.

12.2 Integration into the World Wide Web

The *SEMPER* trials will be based on the World Wide Web. Therefore, we have to define ways of how to integrate *SEMPER* into the existing infrastructure.

12.2.1 Client Side Integration

It is not feasible (man-power and skills) and not even desirable (trial users most likely already possess a browser they are used to and wouldn't like to learn a new one) that *SEMPER* builds its own general -purpose browser. Thus there is a need to integrate the *SEMPER* application in an existing browser.

There are several ways to solve this problem:

1. The *SEMPER* application is launched as a completely independent external MIME-viewer.
2. The *SEMPER* application is launched as an external MIME-viewer and communicates with the browser over a browser-specific client-interface.
3. The *SEMPER* application runs on the local host as a daemon (background process) providing an HTTP-server interface to communicate with the browser.
4. We integrate the *SEMPER* application/library inside a browser such as Netscape which provides a java runtime environment.

Weighting generality (e.g., runs with every browser), portability (does not require fundamental implementation changes when switching platform), flexibility (e.g., does provide an easy way to pass back information to the browser) and ease of

programming (problem of keeping data consistent when multiple *SEMPER* applications run in parallel) we came to the conclusion that approach 3) is the most promising.

12.2.2 Server Side Integration

Integration on the server side does not pose big problems. Most servers already provide an interface (in most cases CGI) which allows simple extension of a server's capabilities by forking a separate process. But as we already implement a minimal HTTP-server on the client side it's more natural to use the same approach on server side thus providing a very uniform environment.

12.3 The *SEMPER* Application

12.3.1 SEMPER Implementation Architecture

The decision taken in the last section leads to a SEMPER implementation architecture which is depicted in the following figure:

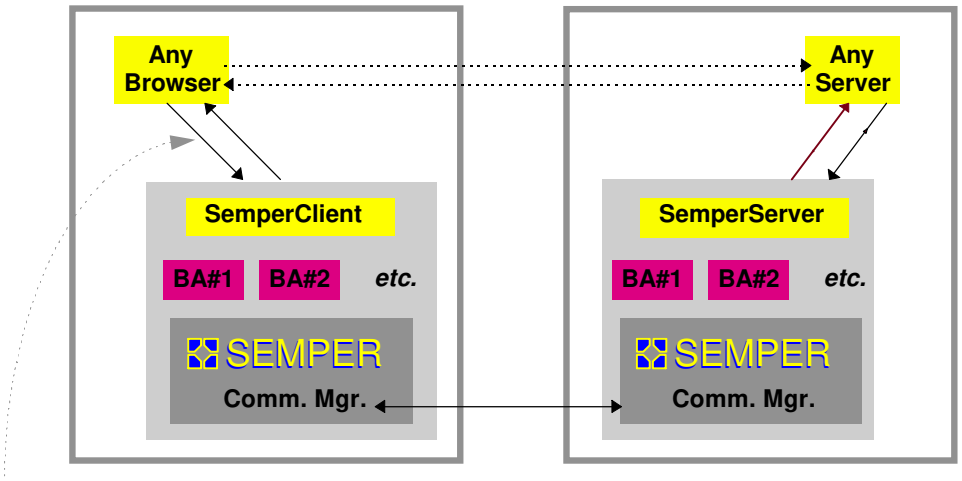


Figure 25: Integration into WWW

The main components of the SEMPER implementation architecture are the following:

- SemperClient
- SemperServer
- Business Application
- SEMPER services

The SemperClient Module

The SemperClient module provides http services to the Business Application framework. It starts an http server and accepts requests from an HTML client. The accepted requests are parsed in order to invoke the generic business application (BA)

with appropriate information objects. The triggered business application is then executed in its own thread. Furthermore, SemperClient caters for the establishment of a communication framework (i.e. by making use of the SEMPER core services) which allows a local Business Application (BA) object and the corresponding BAobject on the server site to communicate. The SemperClient module consists of the following primary elements:

- the scheduler *SCTimer* class
- the monitor *SCMonitor* class
- the listener *SCDaemon* class
- several request parsers *SCHTTParser* class

Class: SCTimer
The main task of this object is to ensure that the duration of a continous execution of a thread object will not exceed a given threashold. Furthermore this object offers some methods that allows a running thread to notify its state. The above state-notification messages are passed to the SCMonitor object.

Class: SCMonitor
The main task of the monitor is to initialise the SCTimer, SCDaemon and several SCHTTParser objects. Furthermore this class offers some methods that process the notification messages coming from the SCTimer object. The most important notification messages are:

- the current job has exhausted the execution duration
- the current job has run into a idle loop
- the current job is proceed an I/O call
- a new client request, via a HTTP browser, has been arrived

Class: SCDaemon
This object is listening to a specific port on the local host in order to process a new client request, made by a HTTP browser to a specific Business Application. When a new client request is accepted, it notifies the SCTimer and stores the new created socket connection object to a common storage place.

Class: SCHTTParser
The following main functionality is provided by this object:

- read the HTTP request that comes from the socket stream
- parse the HTTP request made by the client (via the browser)
- call the BA Session Manager in order to get the requested BA object
- call the Commerce Layer and pass the HTTP request in order to invoke the appropiate BA on the server side.
- pass the new Commerce Layer object to the new local BA
- call the start-up method of the local BA

If no more pending requests are available the SCTimer object will be notified.

The SemperServer Module

The SemperServer module accepts http requests from the SemperClient and initiates the corresponding business application at the server site. The main task of the

SemperServer is to establish an environment where a Business Application (BA) object on the client site and the corresponding local BA object at the server site are able to communicate. The SemperServer module consists of the following primary elements:

- the scheduler *SSTimer* class
- the monitor *SSMonitor* class
- the listener *SSDaemon* class
- several request parsers *SSHTTParser* class
- several BA invokers *SSActionEngine* class

Class: SSTimer
The main task of this object is to ensure that the duration of a continous execution of a thread object will not exceed a given threashold. Furthermore this object offers some methods that allows a running thread to notify its state. The above state-notification messages are passed to the SSMonitor object.

Class: SSMonitor
The main task of the monitor is to initialise the SSTimer, SSDaemon and several SSHTTParser, SSActionEngine objects. Furthermore this class offers some methods that process the notification messages coming from the SSTimer object. The most important notification messages are:

- the current job has exhausted the execution duration
- the current job has run into a idle loop
- the current job is proceed an I/O call
- a new client request, via the Commerce Layer, has been arrived

Class: SSDaemon
This object is layered above the Commerce Layer in order to proceed a new client request for a specific Business Application. For a new client request this object notifies the SSTimer and stores the new created connection object to a common storage place.

Class: SSHTTParser
The following main functionality is provided by this object:

- parsing the HTTP-like request made by the client (via the new created Commerce Layer object)
- call the BA Session Manager in order to get the requsted BA object
- store the parsing results and the BA object to a common storage place in order to be retrieved by a SSActionEngine object

If no more pending requests are available the SSTimer object will be notified.

Class: SSActionEngine
The SSActionEngine object is responsible for the execution of the requested Business Application object. Furthermore the new Commerce Layer object, created from the SSDaemon, will be passed to the new BA in order to allow peer to peer communication between the BA on the client site and the local BA (i.e. at the server site).

12.3.2 A SEMPER Scenario

Initiate a Request

A customer initiates a *SEMPER* business application (BA#1 in the figure) by issuing a *HTTP POST* request with the URL *http://localhost:1234/BA#1* where *1234* stands for a well-known (but yet to be defined) TCP port number. This URL can either be stored in the browsers bookmark-file (e.g. for getting a list of outstanding transactions) or it can be part of a HTML-form which is prepared by the merchant to trigger a business transaction.

The SemperClient takes the HTTP request and initialize a communication channel via the Commerce Layer (CL) to the SemperServer. The connection characteristics of the communication channel are preserved with the creation of a CLConnection object by the commerce layer. After the new connection is established, the HTTP request made by the customer is passed to the SemperServer site. The SemperServer and the SemperClient will parse the HTTP request and raise the appropriate BA locally. In order to allow the message interchange between Client-BA and Server-BA the previously established connection characteristics (i.e. CLConnection object) will be passed to both BAs.

Process a Request

Once started the business application can get all necessary information to initialise itself from the attribute-value pairs of the body of the HTTP POST (e.g. the name of the merchant, a description of the goods to buy etc.). The business application will then call *SEMPER* services to do the desired transactions. A simple application might for example call a `GetOffer` primitive from the commerce layer. The *SEMPER* core will then send this request with the help of the Communication Manager to the merchant. The merchant's previously registered business application (see previous section) will get this request from the *SEMPER* core, handle this request and send the offer back through the *SEMPER* core and the communication manager to the customer. The customers business application can now order the goods by calling `SendOrder` which will be processed in a similar manner. The final result can then be delivered back to the browser as HTTP reply (remember there is still a pending HTTP request).

For business applications which have to be suspended or require additional browser interaction the SemperClient will also allow to resume a transaction. This can be done by specifying a transaction-identifier (tx=23 in our example in the figure above). In that case the SemperClient will not launch a new business transaction but wakes up the suspended transaction and passes all parameters from the HTTP request to it.

12.4 Externalization

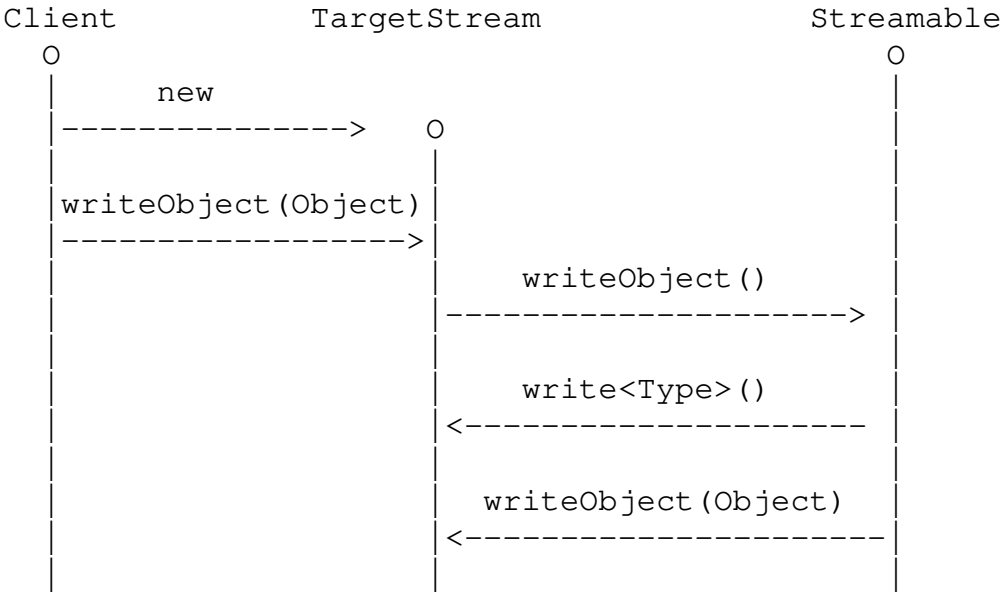
The externalization service defines protocols and conventions for externalizing and internalizing objects. Externalizing an object is to record the object state in a stream of data. Objects which support the appropriate interfaces and whose implementations adhere to the proper conventions can be externalized to a stream (in memory, on a disk file, across the network, etc.) and subsequently be internalized into a new object in the same or a different process. The externalized form of the object can exist for arbitrary

amounts of time, be transported by means outside of the Java process, and can be internalized in a different Java process.

Externalizing and internalizing an object is similar to copying the object. The copy operation creates a new object that is initialized from an existing object. The new object is then available to provide service. Externalization, on the other hand, does not create an object that is initialized from an existing object. Externalization "stops along the way". New objects are not created until the stream is internalized.

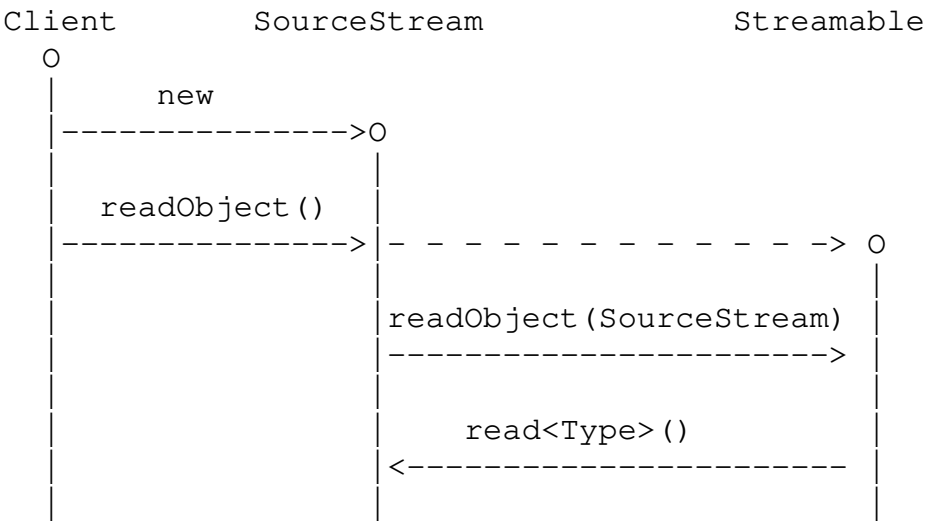
12.4.1 Usage

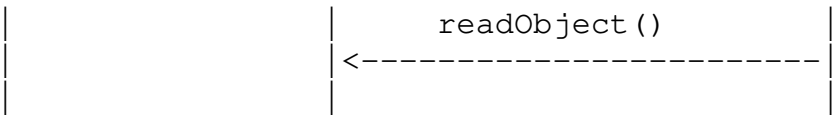
Externalization control flow:



An object implementing the Streamable interface can use the following helper methods of class TargetStream to write scalar values of type = boolean, byte, char, double, float, int, long, short, UTF; e.g. type char has the corresponding method writeChar().

Internalization control flow





The Externalization Service supports following object classes:

- Boolean, Character, Double, Float, Integer, Long, String
- any object class implementing the Streamable interface
- Vector if its elements are as above

Class TargetStream provides a number of class methods to compute the size of the flattened image of an object (in bytes):

- sizeOfBoolean(), sizeOfChar(), sizeOfDouble(), sizeOfFloat(), sizeOfInt(), sizeOfLong(), sizeOfString()
- sizeOfObject(Object o)

Note that method sizeOfObject() only works for objects which are either of class String and Vector, or implement the Streamable interface providing method myStreamSize().

In May 1996, JavaSoft introduced the Object Serialization API for Java. Object Serialization defines interfaces for writing and reading objects, and classes that implement those interfaces. The Object Serialization library will be in the next major JDK release (JDK 1.1), available most probably at the end of 1996. As the functionality of SunSoft's API is almost a proper superset of our own Externalization Service, *SEMPER* plans to replace the Externalization Service class library with SunSoft's Object Serialization. Fortunately, both APIs are close enough such that there should be a smooth transition. If necassary, *SEMPER* will be able to refine the Object Serialization API to its own needs as the source will be available when Object Serialization becomes part of the JDK.

12.5 Routing/Addressing Between Managers and Modules

A flexible architecture requires a general approach for passing objects between managers and modules. The problem is that of addressing and routing. There are two cases: either the two interacting modules are in the same entity and on the adjacent layers, or they are in different entities. Since *SEMPER* architecture is based on the synchronous/blocking model, the first case is solved very naturally. A module requiring the services of the lower layer invokes a method and is blocked until the execution of that method is completed and the result is returned. Note that, especially on a server, this requires multi-threaded coding because the server must be able to handle the requests of many clients concurrently.

The interaction between two modules in two different entities is supported by the communication block. The server opens service-points at ports. A client module uses a port number and a string, referred to as correlator, to establish a channel to a specific server module. After the channel is established the client and server modules can exchange messages. For details, refer to Section 9.1 on the Communication Block. Note that there is a parallel between module interaction within an entity and that across entities: within an entity distinct sessions, i.e. transactions, use distinct threads in order

not to block each other; for the same reason, channels are used, keeping message streams of distinct sessions non-interleaved.

13 Summary and Outlook

This report has described the development and first implementation of the *SEMPER* architecture for electropnic commerce. The layered architecture is designed to support a wide range of electronic commerce applications and payment protocols while anticipating the need to support more complex applications and security requirements in the future. As the report shows, *SEMPER* will use standard libraries and tools as modules to achieve rapid development and platform portability.

The next steps will be to refine the architecture and implement components supporting more advanced business applications. The experience gained by the development and deployment of the first version of *SEMPER* will guide the direction of these extensions.

14 References

[AsSW96] *Asokan, M. Schunter, M. Waidner*: **Optimistic Protocols for Fair hange**, IBM Zurich Research Report RZ 2858 (#90806) 09/02/96.

[DSS] *Digital Signature Standard*. FIPS PUB 186. May 19, 1994.

[EDIFACT] *Electronic Data Interchange for Administration, Commerce and Transport - (EDIFACT) - Application Level Syntac Rules*. Part 5. ISO 9735 -5.

[fast] *Registration Procedures in FAST*. T1.2, deliverable 21.12.1994.

[iso7498] **Information Processing Systems - Open Systems Interconnection - Basic Reference Model;** ISO International Standard 7498; 1989.

[JW96] *P. Janson and M. Waidner:* **Electronic Payment Systems**. Public *SEMPER* report 211ZR018. May 1st 1996.

[LeAn_90] *P. A. Lee, T. Anderson*: **Fault Tolerance - Principles and Practice**; 2nd rev. ed., Dependable Computing and Fault-Tolerant Systems Vol. 3, Springer-Verlag, Wien 1990.

[PAW96] *J. Abad Peiro, N. Asokan and M. Waidner:* **Payment Manager - Overview**. Public *SEMPER* report 212ZR054. March 21, 1996.

[PPSW95] *A. Pfitzmann, B. Pfitzmann, M. Schunter, M. Waidner:* **Mobile User Devices and Security Modules: Design for Trustworthiness;** Univ. Hildesheim / Univ. Dresden / IBM Zurich Research, 1995.

[SHA1] *Secure Hash Standard*. FIPS PUB 180-1. April 17, 1995.

[X509] *Information Technology - Open System Interconnection - The Directory - Part8: Authentication Framework*. CCITT Recommendation X509.

http references (subject to changes):

[CN] *http://www.commerce.net/* (CommerceNet)

[GCT] *http://www.gctec.com/us/Technical/* (GC Tech)

[FSTC] *http://www.fstc.org/* (Financial Services Technology. Consortium)

[FSTCa] *http://www.fstc.org/projects/commerce/public/epaydes.html* (FSTC document “Electronic Payment Infrastructure: Design Considerations”)

[JECF] *http://www.javasoft.com/commerce* (SUN’s JAVA Electronic Commerce Framework)

[JECFa] *http://www.javasoft.com/commerce/doc.white_paper.html* (The JECF White Paper)

[MISF] *http://www.microsoft.com/intdev/security* (Microsoft’s Internet Security Framework)

[OM] *http://www.openmarket.com/* (OpenMarket)

[SDLP] *http://www-diglib.stanford.edu/* (Stanford Digital Library Project)

15 Index

—A—

access control 26; 27
adapter
  payment system 47
arbiter 19; 22
architecture
  ~ framework 13
  model of the real world 18
  SEMPER ~ 24
archive
  ~ service 26
attribute
  security ~ 25
audit 22

—B—

block 3. *See* service block
broker 19
business application
  ~ layer 24
buyer 18

—C—

  24; 29
  ~ manager 32
  ~ service block 25
    24; 29
    24; 29
    24; 29
    24; 29
  24; 30
  ~ authority 30
commerce
  ~ layer 24
  layer 3
communication
  ~ service 25
conditional access 22
container 21; 25; 34
content hosting iv
context 15
contract signing 20
court 20
cryptographic service 25

currency exchange 46
customer *See* buyer

—D—

decision procedure 19; 22
deposit 46
device 13
dialogue 20
directory 19
~ authority 30
~ services iv; 24; 30
disputability 19
dispute 22
~ handling 22
distinguished name 31

—E—

EDI 20
enabling third party 18
entity 13
evidence 19; 22
exception 20; 22
~ handling 22
exchange 20
~ layer 25
exchange layer 36
extending
~ a certificate 31

—F—

fair contract signing 22
fair exchange 20; 22; 36
protocols 38

—G—

generic 14

—I—

information 21
interface 13
external ~ 14
internal ~ 15
layer ~ 13
user ~ 26
invoking entity 13
issuing
~ a certificate 31

—L—

layer 13
business application ~*See* business application layer
commerce ~ *See* commerce layer
exchange ~ *See* exchange layer
SEMPER ~s 24
transfer ~ *See* transfer layer
layer interface 13
loading 46
local 13
long term archiving 20

—M—

mall 20
manager *See* service manager
24; 28
model
~ of the real world 18
module *See* service module
money
as primitive type in transfer 21

—N—

negotiation 15
network 19
non-repudiation 19
notary 20

—P—

payment
~ service block 25
reversal ~ 46
payment block 46
payment model
account-based 47
cash-like 47
payment models 47
payment modules 46
payment system provider 20
peer entity 13
player 13; 18
list of ~s 18
preference 26
primitive type
~ in a container 21
profile 16
protocol 13
provider *See* seller

purse 46

—R—

recovery 16
24; 30
~ authority 30
registration and certification 19
renewing
~ a certificate 31
revoking
~ a certificate 31; 32
role 18

—S—

SAP *See* service access point
seller 18
service 13
~ access point (SAP) 13
~ block 14
~ manager 14
~ module 14
~ primitive 13
~ adapter 15
cryptographic ~ 25
generic ~ 14
generic electronic commerce ~ iv
information ~s 46
manager ~ 15
purse management ~s 46
recovery and dispute management ~s 46
supporting ~ 25
transaction ~s 46
value transfer ~s 46
service manager 15
signed document 21
smartcard 26
24; 28
state 16
statement
~ service block 25
template 24

—T—

24; 28
tamper-resistant memory 26

24
third party
~ accountable 18

distributed trust in ~ 18
enabling 18
trusted (partially) 18
time-stamping 20
TINGUIN 26
transaction 16
state ~ 16
transaction object 15
transfer 20
~ layer 25
~ manager 25
transfer layer 46
transfer manager 46

24; 28
24; 28
trusted interactive user interface *See* TINGUIN
trusted third party *See* third party
TTP *See* third party
type
~ of service 15

—U—

user 18
user interface 26

16 Appendix A: Glossary

Acquiring bank
The bank used by the recipient of electronic money.

Audit trail
Logging information which is the basis for an audit of the use of a service or system. The audit trail may contain non-repudiation tokens to resolve disputes.

Authentication
A proof of identity. This proof can be related to a message being sent meaning that a proof of origin is sent along (see also digital signatures and message authentication codes).

Authorisation
The process of determining the rights associated with a particular principal or role, e.g. rights to access data or services on a computer system.

Business session
A business session is a sequence of commerce layer primitives that operate on the same context, including all case decisions, e.g., for error handling. The primitives could support, e.g., offer, order, invoice and payment.

Business sub-session
A meaningful part of a business session which could itself be called a business session.

Certificate
A certificate is a digitally signed statement about a person. *SEMPER* distinguishes three types of certificates: key certificates linking a public key to a person, attribute certificates linking a particular property to a person and hybrid certificates linking both a key and an attribute to a person.

Conditional access
Access to a service, which is restricted to entities having certain properties or rights.

Confidentiality
Protecting a message against eavesdropping such that it is only meaningful to the intended set of entities.

Container
A data structure used in *SEMPER* to transfer or exchange information and payments. It is a data type structered in the form of a tree. The leaves are nodes specifying or containing (protected) information or payments. Internal nodes contain security attributes.

Context
The context is the local state of an entity. The context evolves from the configuration and preferences set by the user and the results of negotiating parameters with other entities.

Credentials
The personal information provided by a person or institution when registering. (In the literature, the term credential is also sometimes used instead of hybrid certificate).

Customer
The role played by someone wanting to buy something at a merchant.

Decision procedure
A procedure which is used to solve disputes between participants - usually based on audit trails. The decision procedure defines which messages serve as evidence of what and how they are verified.

Digital signature
A digital signature is an electronic counterpart of handwritten signatures. It is verified against a public key, and if the signature system is secure, this serves as a proof that the signed message originates from someone knowing the corresponding secret key

Dispute handler
See Exception handler.

Electronic money
Electronic money is the electronic counterpart to the conventional money.

Entity
Any active element in the *SEMPER* architecture is called an entity. Thus an entity could be a service block, a manager, a module or a collection of these.

Exception handler
An exception handler has to resolve an exception raised by a party. In a first approach the exception handling bases on the assumption that all parties are honest. If this optimistic approach fails, the parties are in a dispute and a more pessimistic approach must be used, usually involving arbiters and finally courts. Exception handling requires all parties to keep sufficient audit trails. For real disputes, the audit trail has to contain evidence, e.g. non-repudiation tokens.

Exchange
An exchange is a protocol whereby a number of parties can exchange business items (payment, information, vouchers, etc.) . In the simplest case these are two transfers.

Fair exchange
A special type of exchange in which each party beforehand specifies what he wants to send and receive. The exchange is fair if the participant is guaranteed to receive an item fulfilling this specification if he sends the promised item.

Hash functions
A function used to compute a fixed length digest of any bit string. Unless stated otherwise a hash function is always assumed to be *collision-resistant*, meaning that it is infeasible in practice to find two different bit string with the same digest.

Identification
A process whereby an entity proves its identity.

Integrity protection
The protection of information against changes by intruders (tampering)

Issuing bank
The bank of the payer in an electronic payment systems.

Non-repudiation tokens
A non-repudiation token is a proof that can later be used as a proof that a certain event took place. These tokens are often necessary in order define a decision procedure. Of

certain interest are **non-repudiation of origin** proving that a message originated from a certain entity and **non-repudiation of delivery** proving that a message has been delivered.

Payment system
Any system that can be used for payments.

Payment system provider
A party (such as a financial institution) that makes a payment system available.

Peer entity
Peer entities are two entities of the same layer but located on different devices. Usually, they interwork to provide a certain service.

Player
A player is a real-world person or body participating in the electronic marketplace, e.g. a buyer, a seller or a third party like a payment system provider, a notary service, etc..

Registration
A procedure by which a participant (usually a real person) registers his or her credentials with a so called registration authority. In return the participant gets a certificate as an electronic proof of the registration.

Revocation
The process of withdrawing the validity of a certificate. A certificate may be revoked by the certification authority. The reason could be that the key is compromised, lost, or was registered by an impostor or that the user abused the granted rights.

Role
The function of an entity in a certain transaction. All transaction protocols are specified for roles, but performed by entities 'playing' the corresponding roles.
Roles are also needed for access control. The access rights usually depend on both the entity and its role.

Security attributes
Security attributes describes the security level that a certain transfer or exchange should provide. Examples are confidentiality, non-repudiation and anonymity.

Service
Services are offered by the joint functionality of a layer to its upper interface. One layer can offer several services, e.g. contract signing or fair exchange of goods against payment are both services of the exchange layer.

Service block
A service block consists of a manager and a number of modules (possibly 0).

Service manager
A service manager provides a common interface to the modules of the service blocks (the *external interface*), plus methods for negotiation and selection of an appropriate module. In addition the service manager provides services providing robustness against fault tolerance. The manager is always part of the *SEMPER* software.

Service module
The service module implements the services provided by the service block. It is expected that modules will be external (i.e., modules will only be implemented as part

of *SEMPER* if necessary). The service module must support the *internal interface* in order to be used by the manager.

Session
A session is a set of primitives that operate on the same context, e.g. a business transaction. Sessions can be nested: a session on a higher layer may start several sessions on lower layers and exchange parts of its context with them.

Sub-transaction
A transaction which is invoked by an ongoing transaction.

Third party
A third party is an entity supporting a business transaction without being involved as business partner.

Transaction
A *SEMPER* transaction is a sequence of method invocations which should result in a state transition from one consistent state into another.

Transfer
The process of transferring any business items (payment, information, vouchers, etc.) from one player to another, usually in a secure way.
A transfer is realized by sending or receiving a container.

Type of module
A service block may support modules implementing the desired functionality in quite different ways and therefore the manager must support different internal interfaces. Modules supporting the same internal interfaces are said to be of the same type.

User
A user is anyone using the marketplace (e.g., buyer or seller).

17 Appendix B: Documentation of the API

The complete documentation of the *SEMPER* application programming interface (*SEMPER*-API) is public and available on-line. This appendix describes how it can be accessed.

The documentation has been generated from the source code using JAVADOC which produces document in the HTML format. To access the documentation, one needs to input the so called URL as “Location” to the browser.

An HTML copy of the overview over the *SEMPER* Architecture (Section 5) with links to the API documentation of all service blocks can be found at URL:

<http://www.semper.org/deliver/d03/javadoc/index.html>.

This overview contains a link to a complete description of all service blocks. Alternatively, this description can be accessed directly at URL:

<http://www.semper.org/deliver/d03/javadoc/packages.html>.